School of Electronic and Electrical Engineering
Faculty of Engineering

UNIVERSITY OF LEEDS

Design and Implementation of an Optimized Computer Vision System for Traffic Sign Recognition and Detection in Autonomous Vehicles; Using a Convolutional Neural Network

Oluwole Oyeniyi Oyetoke

Student ID: *********

Submitted in accordance with the requirements for the degree of
Master of Science
in Embedded Systems Engineering

Supervisor: Dr. David Cowell

Assessor: Dr. Zhiqiang Zhang

The University of Leeds

School of Electronic and Electrical Engineering

August 2017

Declaration of Academic Integrity

The candidate confirms that the work submitted is his/her own, except where work which has formed part of jointly-authored publications has been included. The contribution of the candidate and the other authors to this work has been explicitly indicated in the report. The candidate confirms that appropriate credit has been given within the report where reference has been made to the work of others.

This copy has been supplied on the understanding that no quotation from the report may be published without proper acknowledgement. The candidate, however, confirms his/her consent to the University of Leeds copying and distributing all or part of this work in any forms and using third parties, who might be outside the University, to monitor breaches of regulations, to verify whether this work contains plagiarised material, and for quality assurance purposes.

The candidate confirms that the details of any mitigating circumstances have been submitted to the Student Support Office at the School of Electronic and Electrical Engineering, at the University of Leeds.

Oluwole Oyeniyi Oyetoke

23rd August, 2017

Table of Contents

Abstract

Computers, as we have around today, are GIGO (Garbage-in-Garbage-Out) devices which are only capable of producing results based on what is inputted into them and how they have been originally programmed to respond to such inputs. As such, challenges exist with problem categories that cannot be formulated as algorithms, especially problems which depend on many subtle factors such as knowledge and understanding of previous scenes and corresponding reactions to them. As an example, for the recognition of the Queen of England's image among a cluster of 100 other images, the human brain may be able to provide an informed guess, probably based on past knowledge and various other experiences combined, however, this cannot be accurately derived by a computer without an already pre-written algorithm. In the light of this, there has been growing interest in researches geared toward developing Artificial Intelligent (AI) models which are capable of learning and carrying out classification tasks without making references to any pre-written algorithm. One of such research area is in the field of Neural Networks (NN) which are a biologically inspired family of computation architectures built as extremely simplified models of the human brain [1].

Overall, this project seeks to explore the science behind Neural Networks (NN), its various flavours, application areas and then finally, narrow down by applying it in the design and development of a ***computer vision system which can be used for traffic sign recognition and detection in autonomous vehicles***. The project starts off by designing, developing, implementing and testing a model of the proposed vision system on a CPU using MATLAB and then afterwards, the performance of the implemented vision system is further optimized through vectorization, parallelism, legacy coding and heterogeneous computing. The project is concluded with detailed analysis and evaluation of the various optimization schemes utilized as well as an evaluation of the excellent Neural Network's classification accuracy.

List of Figures

List of Tables

List of Codes

Lists of Abbreviations

2D	Two Dimensional
3D	Three Dimensional
AI	Artificial Intelligent/Intelligence
ALU	Arithmetic and Logic Unit
ANN	Artificial Neural Networks
ART	Adaptive Resonance Theory
ASICs	Application Specific Integrated Circuits
BNN	Biological Neural Networks
BSD	Berkeley Software Distribution
CCA	Connected Component Analysis
CIELAB	French Commission internationale de l'éclairage
CYMK	Cyan, Magenta, Yellow, and Key
CNN	Convolutional Neural Networks
CNS	Central Nervous System
ConvNets	Convolutional Neural Networks
CPU	Central Processing Unit
DAG	Direct Acyclic Graph
DagNN	Direct Acyclic Graph Neural Networks
DBN	Deep Belief Network
DL	Down-left
DM	Down-middle
DNN	Deep Neural Networks
DR	Down-right
DSP	Digital Signal Processing
FCL	Fully Connected Layer
FPGA	Field Programmable Gates Array
GIGO	Garbage in Garbage Out
GTSRB	German Traffic Sign Recognition Benchmark
HDL	Hardware Description Language
HIS	Hue, Intensity and Saturation
HPS	Hard Processor System

HSV	Hue, Saturation and Value
IMDB	Image Database
LRN	Response Normalization Layer
LUT	Look Up Table
MATLAB	Matrix Laboratory
MCP	McCulloch–Pitts
MLP	Multi Layered Perceptron
MSE	Mean Squared Error
MSER	Maximally Stable Extremal Region
NN	Neural Networks
OCR	Optical Character Recognition
OpenCL	Open Computing Language
OpenCV	Open Computer Vision
PCIe	Peripheral Component Interconnect Express
PEs	Processing Elements
PLL	Phase Lock Loops
PNS	Peripheral Nervous System
ReLU	Non-Linearity Unit
RGB	Red, Green and Blue
RNN	Recurrent Neural Network
ROI	Region of Interest
RS	Region Sum
SDK	Software Development Kit
SDRAM	Synchronous Dynamic Random Access Memory
SED	Sobel Edge Detection
SimpleNN	Sequential Neural Network Structure
TL	Top-Left
TM	Top-Middle
TR	Top-Right
YCbCr	Luminance; Chroma: Blue; Chroma: Red

1.0 Introduction

Despite the ever-increasing popularity of transferring (human) tasks to computers for simplification purposes, there are still a lot of human tasks that are still poorly done by computers, such as in areas of visual perception and intelligence [2]. This is because the largest part of the human brain works continuously on data analysis and interpretation while the largest part of the computer is only available for passive data storage. Thus, the brain therefore performs even closer to its theoretical maximum [3]. Although the computer is fast, reliable, unbiased, never tired, consistent and sometimes can even carry out much more complex computational combinations than the human brain is known to muscle, it is still unable to synthesize new rules and it is safe to say, it has no common sense. They rather have a group of arithmetic processing units and storage carefully interconnected to perform complex numerical calculations in a very short time but are not adaptive.

On the other hand, the human brain possesses what we know as common sense, a bigger knowledge base, ability to synthesize new rules and spontaneously detect trends in data without being pre-taught, even though based on capacity, the computer should be more powerful than the human brain as it averagely comprises of over 10^9 transistors with a switching time of 10^{-9} seconds while the brain in comparison consists of over 10^{11} neurons but with only a switching time of about 10^{-3} seconds [3]. With closer analysis, we note that although the human brain is easily tired, bored, biased, inconsistent and cannot be fully trusted, it still outperforms the computer in some application areas due to its perceptive nature of operation (interpretation of sensory information in order to understand the environment). This explains why there still is major reliance on the human brain for classification tasks.

Juxtaposing the computer's strengths and weaknesses against the human brain's makes us realize that in as much as the human brain is better when it comes to perceptive tasks, it has endurance, bias and inconsistency issues. Therefore, effort is being made by researchers to develop systems which are capable of fusing together the advantages of both the brain and the computer into one near perfect outfit. A system which can take on the perceptive learning, out of the box synthesis, self-organizing and self-learning characteristic of the human brain, while maintaining the massive computational capability, speed and enduring features of the computer. This motive has led to increased research on neural networks which are a biologically inspired family of computation architectures built as extremely simplified models of the human brain [1].

This project seeks to design, develop, evaluate and implement a convolutional neural network-based vision system for traffic sign recognition and detection in autonomous vehicle. Figure 1.0 below gives a succinct explanation on the 3 phases (6 stages) this project is divided into.

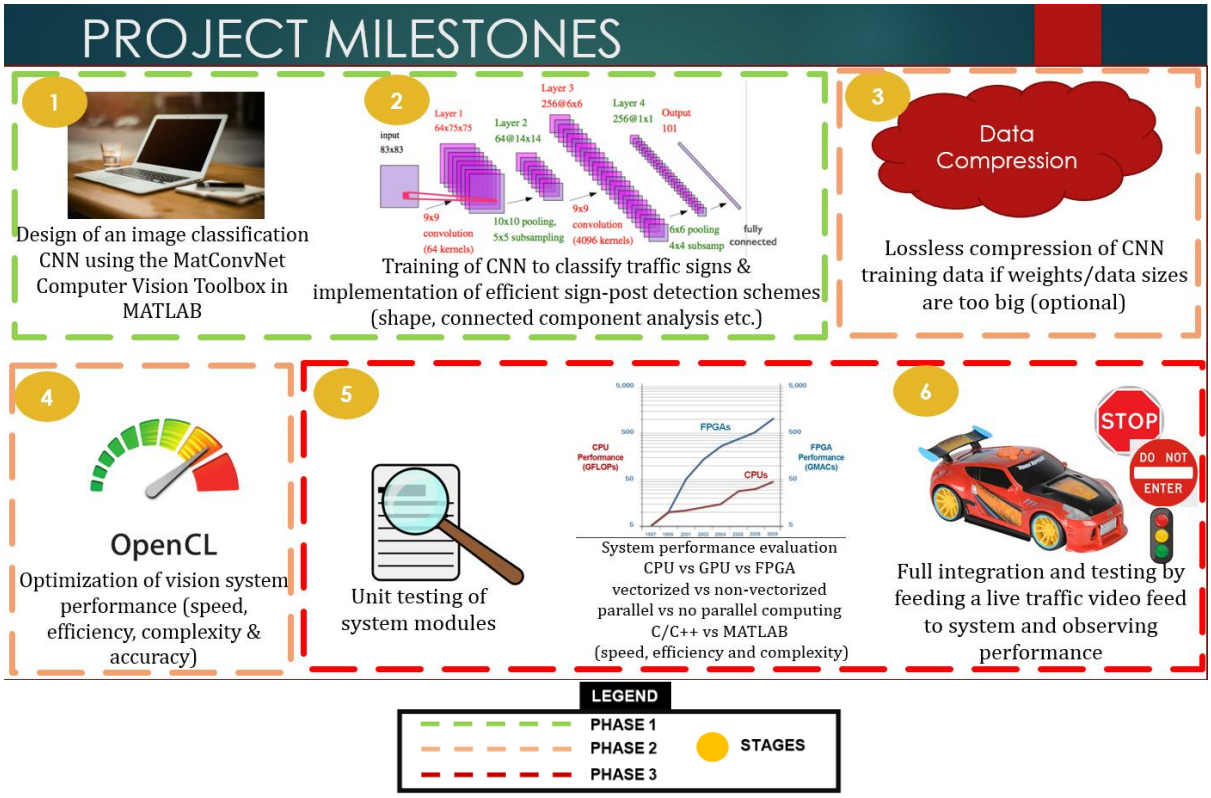

Figure 1.0 Snapshot of Project Mile Stones

1.1 Phase 1

The first phase of this project (Stage 1 and 2) seeks to understand and analyse Neural Networks (NN), its application in deep learning, the fundamentals of the specific NN flavour called the Convolutional Neural Networks (CNN, ConvNets) and thereafter proceeds to practically implement the project's selected ConvNet model called AlexNet on a Central Processing Unit (CPU) for traffic sign classification and detection using the MATLAB MatConvNet toolbox. Also, in this phase of the project efficient traffic sign-post detection modules are designed and implemented.

1.2 Phase 2

The second phase of this project (stage 3 and 4) is aimed at optimizing the performance of the developed system (in phase one), so as to achieve a decent speed of operation at minimal code complexity and still with high efficiency and accuracy.

1.3 Phase 3

The final phase of this project (Stage 5 and 6) evaluates, measures and compares the performance of some of the comprising modules of the vision system on a CPU, Graphics Processing Unit (GPU) and Field Programmable Gates Arrays. Attempting to make justification for the importance of heterogeneous, parallel computing and legacy technologies such as C/C++ and FORTRAN in the development of computationally expensive applications which still have speed as one of their key performance criteria.

The remaining main chapters of this project's report are organised as listed below

- Chapter 2: Background and Concepts (Neural Networks)
- Chapter 3: Examination of The Vision System's Designated Neural Network Model (AlexNet)
- Chapter 4: Design, Implementation, Testing and Training of The Vision System's

2.0 Background and Concepts (Neural Networks)

Neural networks, both in humans and in their artificial replica are made up of interconnected neurons which can pass data between each other and act accordingly on these data. The ANN itself is a near efficient abstraction of the Neural Network of the human body, as they are a biologically inspired family of computation architectures built as extremely simplified models of the human brain [1]. This section describes in detail both the Biological Neural Network (BNN) and the Artificial Neural Network (ANN), juxtaposing their characteristics while also highlighting their fundamental similarities and operating principles.

2.1 Biological Neural Networks (BNN)

The human nervous system can be broadly divided into two main classes which are the Central Nervous System (CNS) and the Peripheral Nervous System (PNS). The CNS consists of the brain and the spinal cord where all the analysis of information takes place, with the brain containing both large scale and small scale anatomical structures carrying out different functions at higher and lower levels. These structures include Molecules and Ions, Synapses, Neuronal microcircuits, Dendritic trees, Neurons, Local circuits and Inter-regional circuit. On the other hand, the peripheral nervous system (PNS) consists of the neurons such as the sensory neurons and motor neurons. The sensory neurons bring signals into the CNS while the motor neurons carry the signals out of the CNS. Based on the different roles and functionality of these neurons, they can be classified under three main classes as explained below.

1. ***Sensor Neurons:*** Sensory neurons get information on activities taking place both within and outside the human body and thereafter brings this information into the CNS so it can be processed.
2. ***Motor Neurons:*** Motor neurons get information from other neurons and convey commands to the human muscles, organs and glands.
3. ***Interneurons:*** Interneurons, which are found only in the CNS, connect one neuron to another. They receive information from other neurons and transmit same to other neurons.

Based on the three classes of neurons, we can categorize their functionality as to receive signals (or information), integrate incoming signals (to determine whether the information should be passed along) and communicate signals to target cells.

Largely, the human nervous system can be broken down into three stages as represented in Figure 2.0 below. The receptors collect information from the environment (e.g. photons on the retina). The effectors generate interactions with the environment (e.g. activate muscles).

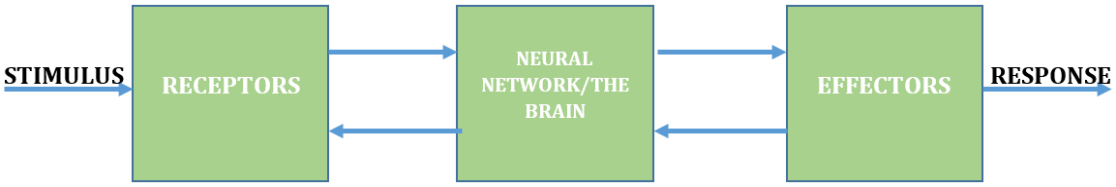

Figure 2.0 Three Stage Human Neural Network Representation

The BNN atomic structure exploited in the realization of Artificial Neural Networks are the Neurons. Figure 2.1 below shows the schematic diagram of a biological neuron and

subsequently, a breakdown of the functions of the various elements that make up the neuron is given.

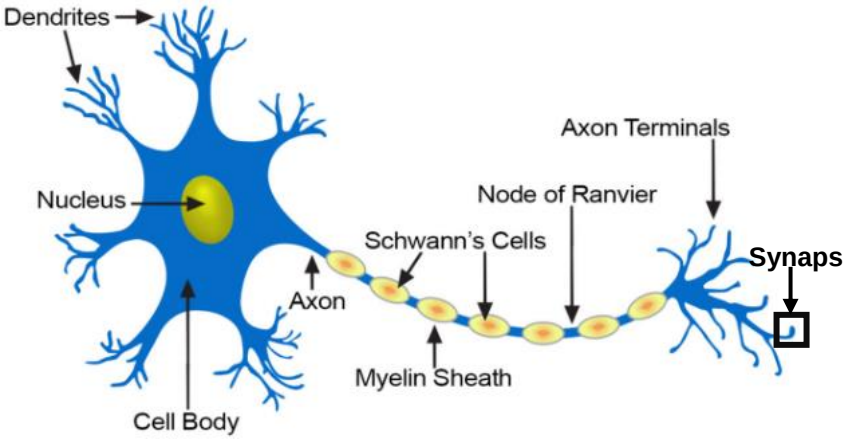

Figure 2.1 Schematic Diagram of a Typical Biological Neuron [4].

The function of the four main parts of the BNN are as listed below

- **Soma:** The soma is the Neuron's cell body. The nucleus of the Neuron resides here and various other extensions such as the Axon and Dendrites are attached to this part of the Neuron.
- **Dendrites:** Branches that receive chemical messages from other neurons. They receive and process incoming information. The interpreted incoming signal can be either excitatory or inhibitory. Whether a neuron is excited into firing an impulse depends on the sum of all the excitatory and inhibitory signals it receives.
- **Axon:** It is basically the trunk of the neuron. If the neuron does end up firing, the nerve impulse, or action potential, is conducted down the axon. Towards its end, the axon splits up into many branches and develops bulbous swellings known as axon terminals (or nerve terminals). These axon terminals make connections on target cells.
- **Synapse:** These are small gaps that exist between the axon of one neuron and the dendrites of the other. Neuron-to-neuron connections are made onto the dendrites and cell bodies of other neurons. These connections, known as synapses, are the sites at which information is carried from the first neuron to the target neuron.

2.1.1 Signal processing in BNN

The Axon-Dendrite connection of the neurons in the BNN do not only suffice for the required transmission of information between them. For adequate transmission of information, the BNN itself has a biological method of signal processing within the network which are based on the steps listed below. The BNN signal processing method is highly dependent on timing as input signals must arrive together and stronger inputs translate to more action potentials per unit time being generated.

1. Signals from connected neurons are collected by the dendrites.
2. The soma sums the incoming signals (spatially and temporally).
3. When sufficient input is received (threshold is exceeded), the neuron generates an action potential.
4. That action potential is transmitted along the axon to other neurons, or to structures outside the nervous systems (e.g., muscles).

5. If sufficient input is not received (threshold not exceeded), the inputs quickly decay and no action potential is generated.

2.2 Artificial Neural Networks (ANN)

Artificial Neural Networks (ANN) are computing system designed to simulate the way the human brain analyses and process information. They are designed to operate using the same/similar principles used by the human Biological Neural Networks for learning and solving classification challenges. Its major elements are its processing unit, topology and learning algorithms. It is important to note that the human brain which the ANN is aimed at being modelled after has over 86 billion interconnected neurons [5] which together form an incredibly complex interacting network enabling humans to see, hear, move, communicate, remember, analyse and understand. Considering the fact that the human brain works in parallel, ANNs are also designed to consists of a bunch of (artificial) neurons that act together in parallel to produce classified outputs, even from previously unknown data. It was developed as a generalization of mathematical models of neural biology based on the following assumptions.

1. Neurons are simple units in a nervous system at which information processing occurs.
2. Incoming information are signals that are passed between neurons through connection links
3. Each connection link has a corresponding weight which multiplies the transmitted signal.
4. Each neuron applies an activation function to its net input which is the sum of weighted input signals to determine the output signal [6].

The first ANN was designed by Warren McCulloch and Walter Pitts in 1943 whose efforts were complemented in 1949 by Donald Hebb, the psychologist at McGill University who developed the first learning law for ANNs. Today, different types of ANN exist, all broadly under the single/multilayer feed forwards, and feed backwards categories.

The Application Areas for ANN include:

1. Pattern Recognition/Classification: Optical Character Recognition (OCR).
2. Biometrics: Speech Recognition.
3. Signal Processing.
4. Control Systems
5. Stock Market Prediction

2.2.1 Fundamental unit of the ANN (the neuron)

Artificial neurons are the constitutive units in an artificial neural network. Depending on the specific model used they may be called a semi-linear unit, Nv neuron, binary neuron, linear threshold function, or McCulloch–Pitts (MCP) neuron [7]. The neuron performs two functions, namely, collection of inputs & generation of an output [8]. Each node output depends only on the information that is locally available to it, at its node, either stored internally or arriving via the weighted connections. As each unit receives inputs from many other nodes, it transmits its output to yet another set of nodes, however, by itself, a single processing element is not very powerful; it generates a scalar output with single numerical value, which is a simple non-linear function of its inputs. The power of the system emerges when layers of these fundamental units (neurons) are cascaded together into a network generally called the Artificial Neural Network. Figure 2.2 below shows a schematic representation of the artificial neuron and its mode of operation.

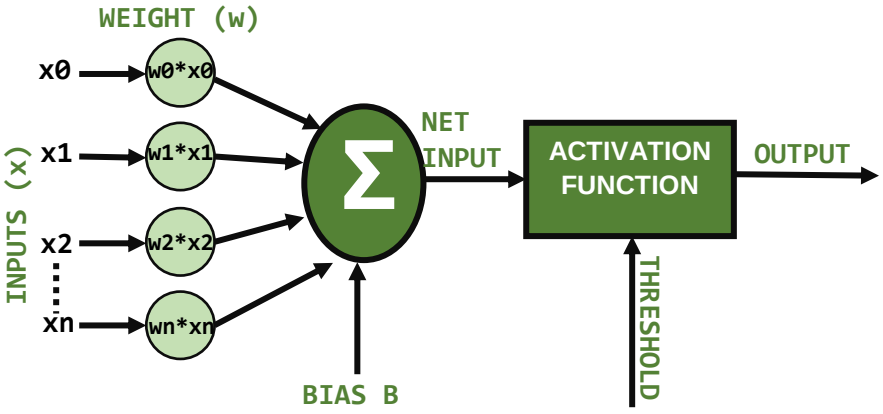

Figure 2.2 Diagram of an ANN neuron

$$Output = Activation\ Function\left(\sum_0^n (x(n) * w(n)) + Biase\right) \quad (1)$$

$$x - Inputs$$

$$w - Weights$$

An initial choice of random weights is given to the connections between the various layers on neurons in the network. Inputs coming in to each of the neurons in the network are multiplied by the respective weights of their connection path while the neuron sums up these inputs with its own bias and then passes the net input through a specified activation function which has a threshold value. If the result of the computation performed by the activation function exceeds the threshold value, an output is triggered, otherwise, no output is triggered. This is analogous to the model of signal processing used by the BNN. It is also important to note that these weight values changes as the network gets tuned over time.

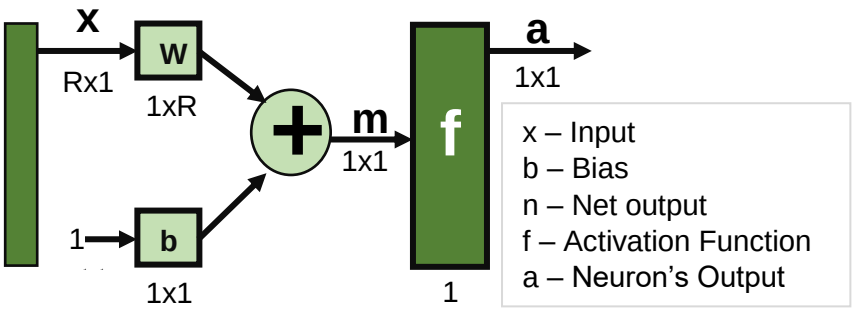

Figure 2.3 Abbreviated (Matrix) Notation for Neuron with Multiple-Input

$$m = \sum_0^n \left(\begin{bmatrix} x0 \\ x1 \\ x2 \\ x3 \\ . \\ xn \end{bmatrix} * [w0\ w1\ w2\ w3\ \ldots . wm] \right) + Biase \quad (2)$$

$$f(m) = a \quad (3)$$

2.2.2 ANN activation functions

In biologically inspired neural networks, the activation function is usually an abstraction representing the rate of action potential firing in the cell. For the ANN, a feature is added to

serve as the decision maker which finally maps the sets of input to a particular space in the output domain. This characteristic is called the activation function (or transfer function) which when applied to the net input of the ANN neuron produces a specified output. It limits the amplitude of the output of neuron to some finite value [8]. An acceptable range of output is usually between 0 and 1, or -1 and 1. Many activation functions have been tested for artificial neurons but only a few have found practical application. As we know, the individual spike timings are often important, which makes “spike time coding” the most realistic representation for artificial neural networks [9].

2.2.2.1 Identity function

This function provides an output which is equal to the neuron weighted input. It is also called linear function. Neurons with this function are used for the input units of ANN and for linear approximation. It is represented as stated in (4) below.

$$F(z) = z \tag{4}$$

$$Z = sum\ of\ weighted\ input\ into\ the\ neuron$$

$$Range = (-\infty, \infty)$$

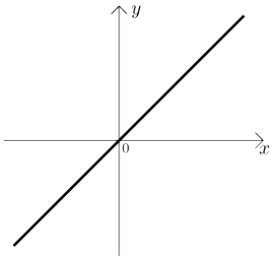

Figure 2.4 Identity Function

2.2.2.2 Step functions

They are also called the hard limit function They are often used in decision making neurons for classification and pattern recognition tasks. There are two types of step functions:

- *Binary step function:* This is also known as Threshold or Heaviside function. It is represented as in (5) below.

$$f(z) = \begin{cases} 1\ if\ z \geq T \\ 0\ if\ z < T \end{cases} \tag{5}$$

$$Range = (0,1)$$

$$\text{where z} = \text{sum of weighted inputs into the neuron}$$

$$\text{T} = \text{threshold}$$

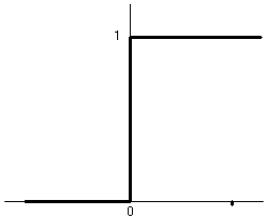

Figure 2.5 Binary Step Function

- Bipolar step function: This is also called sign function. It is represented as in (6) below

$$f(z) = \begin{cases} 1 \; if \; z \geq T \\ -1 \; if \; z < T \end{cases} \quad (6)$$

$$Range = (-1,1)$$

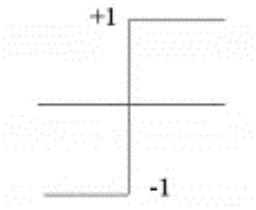

Figure 2.6 Bipolar Step Function

2.2.2.3 Sigmoid function

These are S-shaped curves that are very useful in multi-layer networks. The two types are:

- *Binary sigmoid function*: This is also called logistic function. It is a sigmoid function with range from 0 to 1 and used for neural networks in which the desired output is either binary or are in the interval between 0 and 1. It is represented as in (7) below

$$f(z) = \frac{1}{1+\exp(-\sigma z)} \quad (7)$$

$$where \;\; z = \;\; sum \; of \; weighted \; inputs \; and \; \sigma \; = \; steepness \; parameter$$

$$Range = (0,1)$$

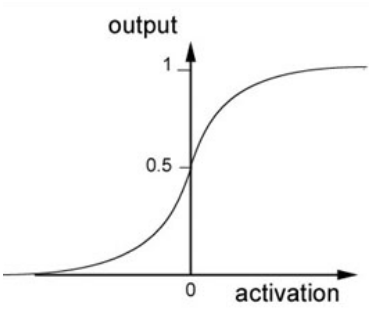

Figure 2.7 Binary Sigmoid Function

- *Bipolar sigmoid function:* The range of output for this function is from -1 to 1. It is closely related to the hyperbolic tangent function. It can be used as the activation function when the desired output range is between -1 and 1. It is generically represented as:

$$f(z) = \frac{1-\exp(-\sigma z)}{1+\exp(-\sigma z)} \quad (8)$$

While the hyperbolic tangent is represented as:

$$h(x) = \frac{1-\exp(-2x)}{1+\exp(-2x)} \quad (9)$$

$$Range = (-1,1)$$

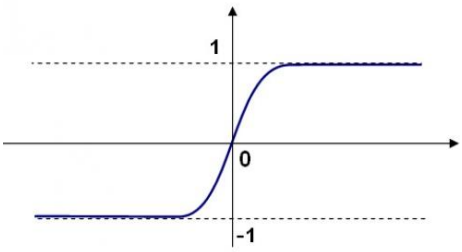

Figure 2.8 Bipolar Sigmoid Function

2.2.2.4 Hyperbolic tan

This is represented by (10) below

$$f(z) = \tanh(z) = \frac{2}{1+e^{-2z}} - 1 \tag{10}$$

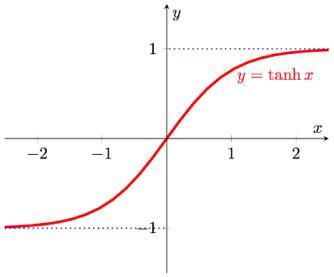

Figure 2.9 Hyperbolic Tan Function

2.2.2.5 Rectified Linear Unit (ReLU)

This is represented mathematically by (11) below

$$f(z) = \begin{cases} 0 \; if \; z < 0 \\ z \; if \; z \geq 0 \end{cases} \tag{11}$$

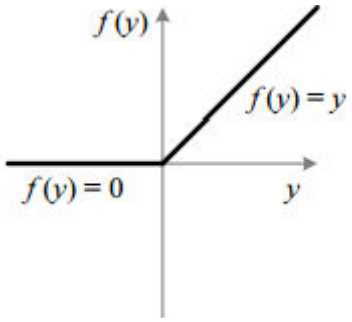

Figure 2.10 Rectified Linear Function

2.2.3 ANN training mechanisms overview

The ANN neurons are connected to form a computational network. Once the desired network has been structured for an application, it is proceeded to the training stage which starts with the initial choice of random weights for each neurons connection inlet (weighted connection). These initial random weights are adapted over the training process to produce a finally suitable weight value which on interaction with future input data will most likely produce the desired results even for previously unknown data. In other words, in the training process, the network is stimulated by its environment (through sets of input data) Due to the stimulation, the network experiences some changes in its internal parameters (weights) as regulated by a

group of pre-written rules. These changes to the internal structure makes the network responds to its surroundings in a different way on future interactions. The pre-written rules defined to solve the learning process of the ANN are categorized under different classes which are:

- Supervised Learning (Error Based)
- Unsupervised Learning
- Reinforcement Learning

2.2.3.1 Supervised learning

The scope of this project covers the use of neural networks developed through a supervised learning process, and as the name indicates, supervised learning is a method of training artificial intelligent systems by feeding them with data along sides information. In other words, training data includes both the input and the desired results (supervisory signal) [10]. Essentially, an effective enough training process leaves the system at a state whereby it can confidently classify further inputs to the system based on its previous training experience. The network processes the training inputs and compares its resulting outputs against the desired outputs and errors are then propagated back through the System (back propagation) causing the system to adjust the weights which control the network. This process occurs over and over as the weights are continually adjusted until decent weight values are generated capable of producing near correct classifications when fed with raw data in the future. There exist two patterns of implementing supervised learning on an ANN which are as listed below.

- Stochastic or Online Training
- Batch or Offline Training

In the online training method, the network weights are adjusted after each pattern presentation. The next input pattern is selected randomly from the training set to prevent any bias that may occur due to the sequences in which the patterns occur in the training set. The weights vary dynamically as new information is entered into the system while in the offline training method, the network weight changes are accumulated and used to adjust weights only after all training patterns have been presented. Here, the ANN is distinguished between training phase and operation phase. In the training phase, a group of data called training samples are is used. The training sample contains examples from where the network will take the knowledge. This project uses the online training pattern in two phases.

- *Phase 1(Propagation Phase):* Essentially, in this phase, the forward propagation of the training inputs is made across the neural network in order to generate the network's output value(s). After this, a back-propagation process for error correction is engaged in.
- *Phase 2 (Weight update):* The percentage contribution of each weight to the network’s total error value will be calculated and the result will be used in determining how to adjust each of the weights so as to minimize the total error of the output of the system.

Phases 1 and 2 are repeated until the performance of the network is satisfactory. It is important to note that the speed at which the neural network learns is highly dependent on the chosen learning rate for the system

2.2.3.2 Unsupervised learning

ANNs are now also applicable for unsupervised operations whereby the network must make sense of the inputs without outside help (self-organization or adaption). In this type of learning, the network learns about the pattern from the data itself without a priori knowledge and the network basically performs a clustering of the input space. One of the most important

features in neural networks is its learning ability, which makes it in general suitable for computational applications whose structures are relatively unknown [11]. However, neural networks in unsupervised learning are widely used to learn better representations of the input. Among neural network models, the self-organizing map (SOM) and adaptive resonance theory (ART) are commonly used unsupervised learning algorithms. The SOM is a topographic organization in which nearby locations in the map represent inputs with similar properties. SOM represent types of NNs that have a set of neurons connected to form topological grid (usually rectangular). When some pattern is presented to SOM, neuron with closest weight vector are considered winners and their weights are adapted to the weights of their neighbourhood. This way SOM naturally finds data clusters. The ART model allows the number of clusters to vary with problem size and lets the user control the degree of similarity between members of the same clusters by means of a user-defined constant called the vigilance parameter. ART networks are also used for many pattern recognition tasks, such as automatic target recognition and seismic signal processing [12].

2.2.3.3 Reinforcement learning

The reinforcement learning class is a bit more interesting, as, although a teacher is present, it does not present the expected answer but only indicates if the computed output is correct or incorrect. The information provided by the “teacher” helps the network in the learning process.

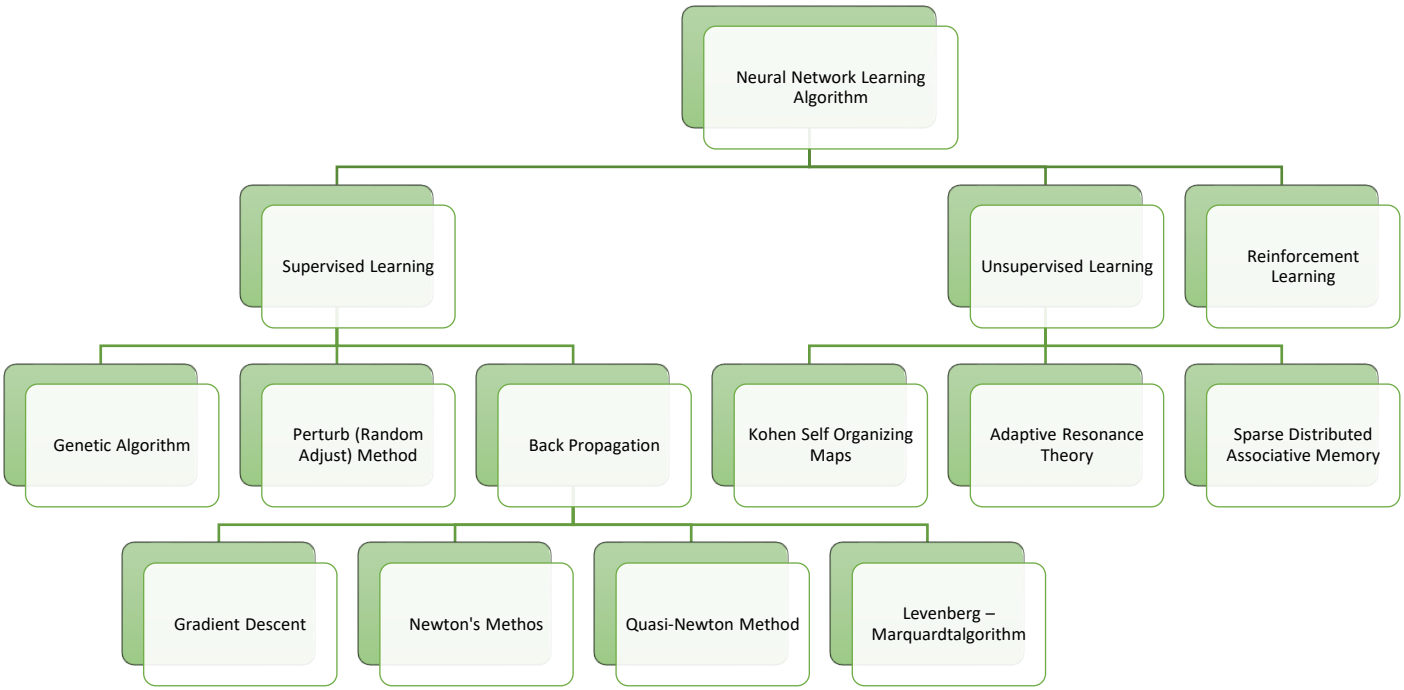

Figure 2.11 Hierarchical Model of the ANN Learning Class and Algorithms

2.2.4 ANN back propagation and weight update philosophy

The backward propagation of errors or backpropagation, is a common method of training artificial neural networks and used in conjunction with an optimization method such as gradient descent. After the pioneering work of Rosenblatt and others, no efficient learning algorithm for multilayer or arbitrary feedforward neural networks was known. This led some to the premature conclusion that the whole field of ANN had reached a dead-end. The rediscovery of the backpropagation algorithm in the 1980s, together with the development of alternative network topologies, led to the intense outburst of activity which put neural computing back into the mainstream of computer science [13]. The backpropagation algorithm was a major milestone

in machine learning because, before it was discovered, optimization methods were extremely unsatisfactory. One popular method was to perturb (adjust) the weights in a random, uninformed direction (increase or decrease) and see if the performance of the ANN improved. If it did not, one would attempt to either go in the other direction, educe the perturbation size or a combination of both. Another attempt was to use Genetic Algorithms to evolve the performance of the neural networks. Unfortunately, in both cases, without (analytically) being informed on the correct direction, results and efficiency were suboptimal [14]. This was where the backpropagation algorithm came into play by repeating a two-phase cycle (propagation and weight update) through which the network is able to learn on its own with less human trials and error approaches. Essentially, the two phases of the back propagation can be intuitively viewed as the steps below.

1. Feed-forward computation
2. Back propagation to the output layer
3. Back propagation to the hidden layer
4. Weight updates [15]

When input data is presented to a Neural Network, it is propagated layer-by-layer through the network up-to the output layer. This is called forward propagation. Using a cost function, the output of the network is then compared to the desired output and an error value is calculated for each of the neurons in the output layer. The error values are then propagated backwards, and used to adjust the weight value of each of the neurons which make up the layers of the neural network. Backpropagation uses these error values (from each of the output neurons) to compute the gradient of the cost function with respect to all the weights in the network. With this, as the network is trained, the neurons in the intermediate layers organize themselves in such a way that the different neurons learn to recognize different characteristics of the total input space. After the training process, when a previously unseen input pattern is presented to the network, neurons in the hidden layer of the network will respond with an active output if the input contains already learned features. It is important to note that backpropagation requires a known output for every input value to the network in order to calculate the cost function's gradient. This explains why it is usually referred to as a supervised learning method.

Figure 2.13 below shows a graphical representation of the gradient descent operation on the cost function during the back-propagation process. In summary, with the backpropagation methodology, the loss function calculates the difference between the output of the network after being fed with input training samples against the network's expected output. Note that these outputs are numeric values and are mainly dependent on the various mathematical computations within the various layers of the NN.

2.4.4.1 ANN weight update computation (cost function)

In the ANN (supervised learning), the output of the network's computation is compared to the desired output and by using a desired cost (loss) function an error value is calculated for each of the neurons in the output layer. There are different kinds of cost functions that can be used, some of which are listed below

- Mean Squared Error (MSE) Cost Function
- Cross Entropy Cost Function
- Kullback-Leibler Divergence
- Exponential Cost
- Hellinger Distance
- Itakura–Saito distance

The cost function used in this project is the MSE.

Cost function as the name implies are functions which help us determine the level of divergence an output is compared to a desired output, consequently helping us draw a line of best fit through a stream of data. The cost function on its own is a mathematical hypothesis which minimizes the error between a set of given input and outputs. (12) below shows the hypothesis formula for a linear regression with one variable which predicts that output y is a linear function of input x.

$$h_0(x) = \theta_0 + \theta_1 x \tag{12}$$

$$Where\ h_0(x) = Hypothesis$$

$$\theta_i = parameters\ of\ the\ model$$

$$x = input\ to\ the\ model$$

(13) below describes the hypothesis for data set with more than one input variable,

$$h_0(x^i) = \theta_0 + \theta_1 x^i \tag{13}$$

The cost function seeks to find a valid set of θ so that $h_0(x)$ is close to output y for every piece of input x supplied to the system i.e. in a supervised learning environment. Therefore, minimizing the error between the hypothesis and the reality by choosing suitable θ. This can be achieved by (14) below

$$\min_{\theta_0\theta_1} = J(\theta_0, \theta_1) = \frac{1}{2}\sum_{i=1}^{m}(h_\theta(x^i) - y^i)^2 \tag{14}$$

$$M = no\ of\ training\ set$$

Thinking intuitively through the algorithm of this function, different values of θ which make up the hypothesis are chosen and tested in equation 14 above to derive different cost values. This explains why it is also called the squared error function. This square isn't there for no reason, as it allows the result to be quadratic. We know that a quadratic function when plotted will always have a visible 'u' shape making it convex, having only one minimum, unlike non-quadratic graphs which could have local and global minimums [16]. This feature will come in handy should the gradient descent algorithm need to be used. For the ANN, the hypothesis h_θ and output y are represented by the actual output of the Neural Network (NN) and the targeted output of the network. The entire error calculation for one sweep of training data across the network (1-Epoch) is realized by analysing the error/difference at the output nodes. i.e. comparing the difference in the networks output value for each of the output node(s) n against the targeted/expected output. This error is mathematically represented by the (15) below

$$E_p = \frac{1}{2}\sum_{1}^{n}(T_n - N_n)^2 \tag{15}$$

$$N_n - network\ values$$

$$T_n - target\ values$$

P represents the sweep number e.g. at p=1, this represents the first sweep of training data across the NN. In other words, to derive the total error over the entire training iterations, equation r below can be used

$$E = \sum_{1}^{p} E_p = \frac{1}{2}\sum_{1}^{p}\sum_{1}^{n}(T_n^p - N_n^p)^2 \tag{16}$$

E is total error, and p represents all forward propagation iteration and n = number of output nodes. A normalized version of the total error is given by the MSE in (17) below

$$E = \sum_1^p E_p = \frac{1}{2pn}\sum_1^p \sum_1^n (N_n^p - T_n^p)^2 \quad (17)$$

By computing the partial derivative of the total loss with respect to the individual weights at the connection between each of the individual neurons that make up the system, we can obtain how much a change in that weight value affects the total error E. The goal of backpropagation is to update each of the weights in the network so that they cause the actual output to be closer the target output, thereby minimizing the error for each output neuron and the network as a whole [17]. However, to get the most minimal error, it is wise to also get the most minimum value from all the possible cost functions generated for a set of data. This can be gotten by using gradient descent which is a more general algorithm which can be used to minimize cost functions and even other functions. There exist also other algorithms which can be used in place of the gradient descent which are

- *Newton's Method*: The objective of this method is to find better training directions by using the second derivatives of the loss function [18].
- *Quasi-Newton Method:* This method build up an approximation to only information on the first derivatives of the loss function [18].
- *Conjugate Gradient:* The conjugate gradient method can be regarded as something intermediate between gradient descent and Newton's method. Here, the training directions are conjugated with respect to the used Hessian matrix [18].
- *Levenberg – Marquardt algorithm:* Also known as the damped least-squares method. It works with the gradient vector and the Jacobian matrix.

The slowest training algorithm is usually gradient descent, but it is the one requiring less memory. On the contrary, the fastest one might be the Levenberg-Marquardt algorithm, but usually requires a lot of memory. A good compromise might be the quasi-Newton method [18].

Using the gradient descent, maximally minimizes the total networks output error, every time we want to update our weights, we subtract the derivative of the cost function w.r.t. the weight itself, scaled by a learning rate from the current weight. In the case where there is more than one neural connection to the node to deal with, it means the weight values have to be simultaneously updated.

$$w := w - \alpha \frac{\delta E}{\delta w} \quad (18)$$

$$Where\ w = weight$$

$$E = cost\ function$$

$$\alpha = selected\ learning\ rate$$

Figure 2.12 Figure Used to Explain The Gradient Descent Operation Per Neuron

For an arbitrary node as in Figure 12 above which has an input connection weight w, net input 'in' an output of 'o', with an expected target output of t and constitutes the part of a network with an MSE of E, the deferential of the MSE with respect to its input connection weight w is given by (19) below based on chains rule

$$\frac{\delta E}{\delta w} = \frac{\delta E}{\delta o} * \frac{\delta o}{\delta in} * \frac{\delta in}{\delta w} \quad (19)$$

In words, this can be read out as:

'*How much does the total error change with respect to the output * How much does the output 'o' change with respect to its total net input * how much does the total net input of in change with respect to the input weight w'*

By using the quotient rule given in (20) below, the individual differentials in equation t above are reduced as described in (21), (22) and (23) below

$$d\left\{\frac{f(x)}{g(x)}\right\} = \frac{g(x)f^{|}(x) - f(x)g^{|}(x)}{(g(x))^2} \quad (20)$$

$$\frac{\delta E}{\delta o} = -(t - o) \quad (21)$$

$$\frac{\delta o}{\delta in} = o(1 - o) \quad (22)$$

$$\frac{\delta in}{\delta w} = in \quad (23)$$

(19) above can be re-written as (24) below

$$\frac{\delta E}{\delta w} = -(t - o) * o(1 - o) * in \quad (24)$$

Therefore to decrement the error and adjust the weight for every node, the new weight w(+1) is given by (25) below

$$w(+1) = w - \alpha(-(t - o) * o(1 - 0) * in) \quad (25)$$

Once a complete back propagation has been carried out and the weight adjusted, a new sample is fed to the network and its output is supervised. Effort is made to correct errors detected using the same back propagation technique until the near perfect classifier is generated. As we know, a supervised learning algorithm analyses the training data and produces an inferred function, which is called a classifier. This inferred function (made up of the network's weights) should predict the correct output value for any valid input object.

By iterating through the network multiple times and minimizing the error effect of each input weight on the total error of the network, the learning algorithm gradually descends towards the least possible error value until convergence is reached. The learning rate controls how steep/fast the function steps down the gradients surface. It might be very difficult for the system to converge when the learning rate chosen is too large, however, on the other hand, choosing an extremely minimal learning rate would make the system take longer time to converge.

2.4.4.2 Intuitive analysis of the gradient descent operation

As was highlighted in the sections above, the MSE formulae is designed with a squared in it (which is later cancelled out by the multiplication by 0.5) so that the result of the function is quadratic. It is known that a quadratic function when plotted will always have a visible 'u' shape making it convex, having only one minimum, unlike non-quadratic graphs which could have local and global minimums [19]. As the gradient descent function adjust the weight values through the network, it systematically converges towards a set of weight values which will generate the least error difference. Note that the least error difference graphically can be found on the points corresponding to the lowest point of the 'u' shaped graph. What the derivate in the gradient descent formulae does is simply pick a point on the MSE plot based on the

selected weight value and look for the slope of the line tangent to that point. In other words, (18) above can be re-written as (26) below.

$$w := w - \alpha * (positive\ slope) \quad (26)$$

Therefore, at every iteration, the value of w reduces towards the minimum. At the local/global minimum, the derived slope will be zero, therefore bringing the system to convergence. There is no need to decrease the learning rate over time, because as one approaches the local minimum, the slope approaches 0 and the step becomes smaller.

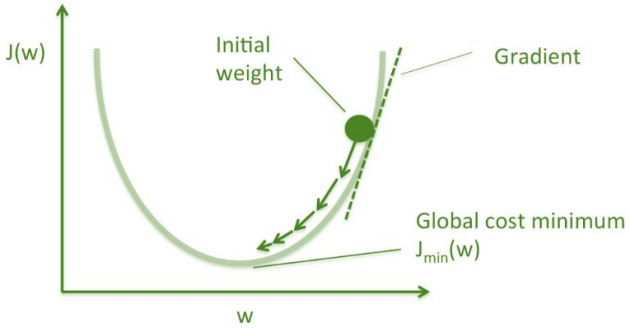

Figure 2.13 Convergence of MSE Function Through Gradient Descent [20]

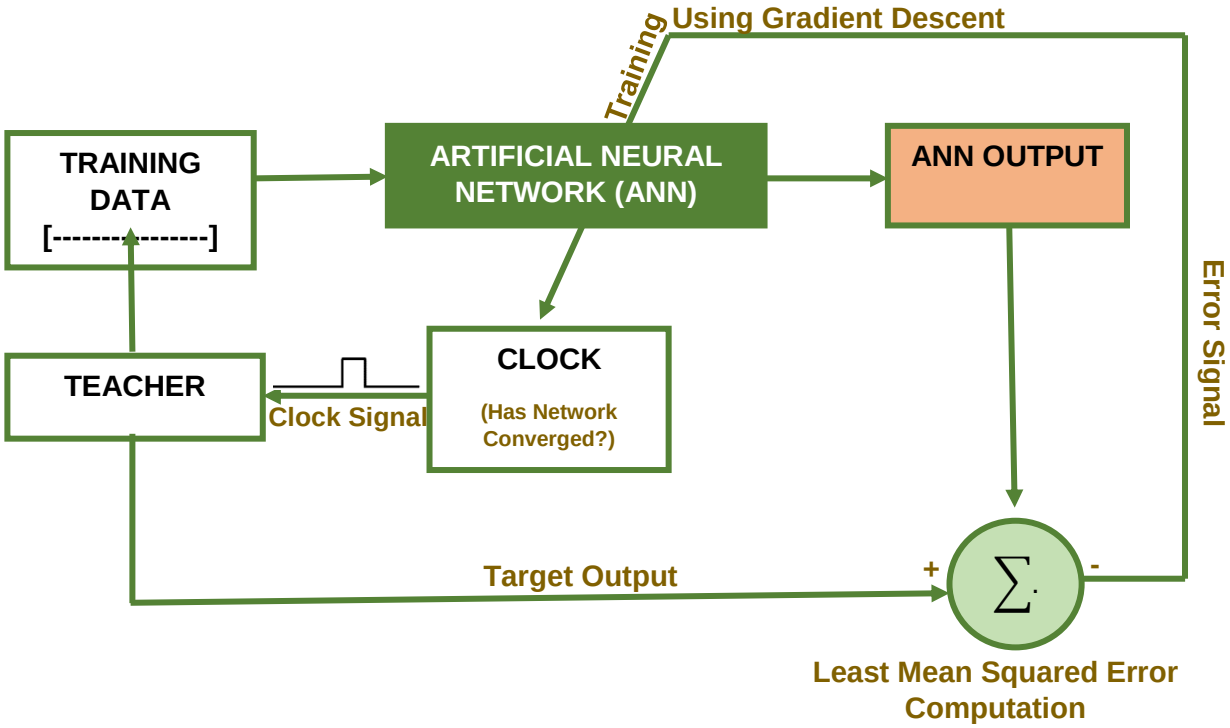

Figure 2.14 Supervised Learning Process for the Neural Network

It is important to note that there are many laws (algorithms) used to implement the adaptive feedback required to adjust the weights during training. Backpropagation just happens to be one of the most commonly used and known. Training the ANN requires a conscious analysis, to ensure that the network is not over trained and when finally, the system has been correctly trained, and no further learning is needed, the weights can, if desired, be "frozen." In some systems, this finalized network is then turned into hardware so that it can be fast. Other systems don't lock themselves in but continue to learn while in production use.

2.2.5 ANN Architecture

The neural networks topology is made up multiple layers of individual units of neurons acting together in a distributed environment. The architecture is primarily made up of an input layer, series of hidden middle layer(s) and then an output layer. Figure 16 below gives a visual description of a typical ANN architecture.

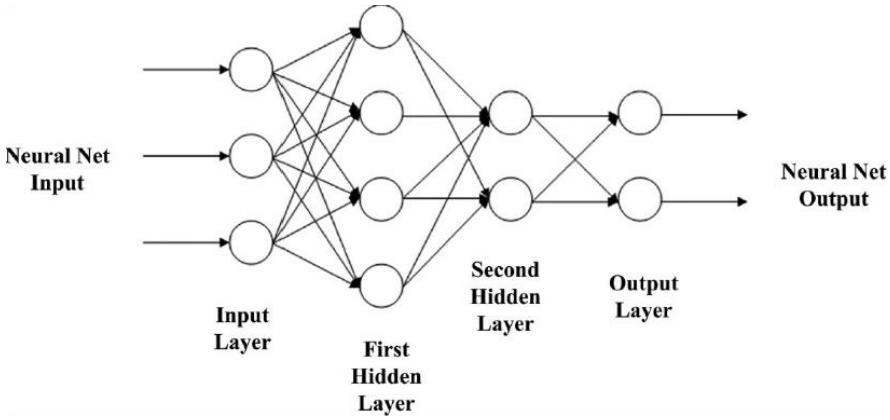

Figure 2.15 Typical Artificial Neural Network Architecture [21]

Estimating the number of neurons and hidden layers that should make up a neural network for a particular application remains a difficult task. Research has shown that the more internal structure a network has, the better that network will be at representing complex solutions. On the other hand, too much internal structure makes the network slower and can even cause training to diverge, or lead to overfitting which would prevent the network from generalizing well to new data (Generalization: how well a network model performs on unseen data).

The size of the input to the network should correspond to the size of the input vector, i.e. the parameters which are to be analysed in parallel to generate the output (answer). Similarly, the size of the network's output layer should correspond to the number of answers the network is designed to feed out. However, the internal/hidden layers can be structured in varying ways according to the intuition and taste of the network developer. Efforts have been made to define guiding rules which would help in ANN topology design, however, these rules have not proven to be a standard yet. Few of the approaches taken today in designing an ANN topology aimed at being used for an Artificial Intelligent application are listed below

- *Trial and Adjust Method*: Here the developer tries different configurations and evaluates the networks performance in each case and there after selects the best performing architecture.
- *Rule of Thumb:* Some schools of thoughts have suggested some best practices concerning the number of neurons in the hidden layer. For examples, there have been speculations that the number of neurons in the hidden layers should be between the input and output layer size, never larger than twice the size of the input layer, about the two-thirds of the sum of the input and output layer size and various other such suggestions
- *Dynamically Adjusting Algorithms:* Algorithms such as 'Cascade Correlation' start with a very simple network and then over during training, add hidden nodes.

Results from research has shown that in certain cases, training can be more successful in larger networks, and consequently, it is possible for larger networks to result in improved generalization [22].

3.0 Examination of The Vision System's Designated Neural Network Model (AlexNet)

In 1989, LeCun et al. introduced Convolutional Neural Networks (ConvNets, CNN) for application in computer vision [23]. Convolutional Neural Networks use images directly as input, and instead of handcrafted features, Convolutional Neural Networks are used to automatically learn a hierarchy of features which can then be used for classification purposes. This is accomplished by successively convolving the input image with learned filters to build up a hierarchy of feature maps. The hierarchical approach allows to learn more complex translation and distortion invariant features in higher layers. Deep Convolutional Neural Networks are analogous to traditional ANNs and can be trained more easily using traditional methods such as a combination of parameter optimization and error back propagation, just like is obtainable in conventional ANNs. This property is due to the constrained architecture of convolutional neural networks which is specific to input for which discrete convolution is define [24]. The original Convolutional Neural Network is based on weight sharing [25] and the only notable difference between CNNs and traditional ANNs is that CNNs are primarily used in the field of pattern recognition within images. This allows developers to encode image-specific features into the architecture, making the network more suited for image-focused tasks whilst further reducing the parameters required to set up the model [26].

As we know, Neural Networks are biologically inspired, and just like the Multi Layered Perceptron (MLP) ANN, the connectivity pattern between the neurons of a CNN is also biologically inspired, but by another kind of biological organization/operation which in this case is related to the organization of the animal visual cortex which has small regions of cells that are sensitive to specific regions of the visual fields. It was discovered that all the neuronal cells were organized in a columnar architecture and that together, they could produce visual perception. In summary, the observation showed that some of the neurons in the brain responded only in the presence of edges of a particular orientation, and this idea of having a system made up of components which are only responsive to specific features later became the founding idea behind the CNN.

Overall, the CNN takes in a collection of inputs and processes these inputs through a series of convolutional stages with different classes of filters which make up its different layers in order to extract specific features from the input. The extracted specific features are then passed on to the Fully Connected Layer (FCL) of the architecture which is primarily a classifier. CNNs are particularly used for image classification, especially in the case were huge data sets are involved. As we know, Image classification is the task of taking an input image and outputting a class (a table, chair, stool) or a probability of classes that best describes the image. The CNN takes an input array representing an image (coloured or grey scale) and produces an output numbers that describe the probability of the image being a certain class. As can be seen from section 3.2.1 and 3.2.3 of this report, the Convolution and Pooling layers act as Feature Extractors for the network while the Fully Connected layer acts as a Classifier.

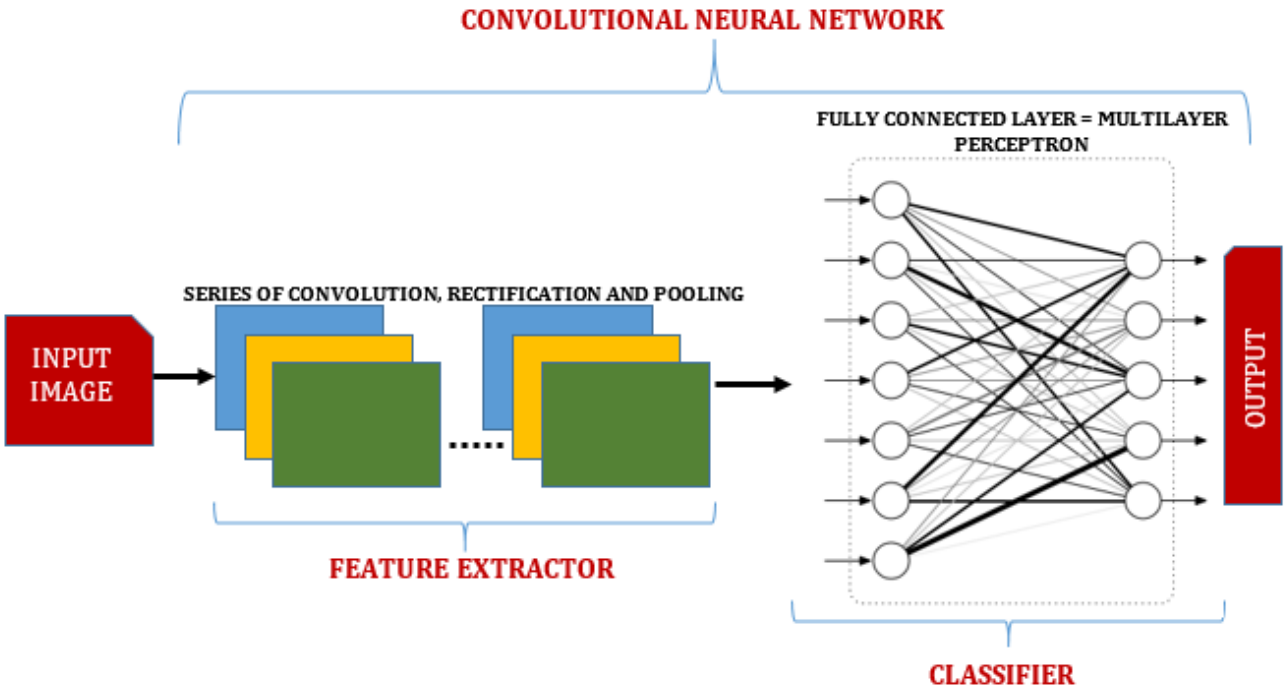

Figure 3.0 High Level View of the CNN

CNN represents one out of the various deep learning architectures such as the Deep Neural Networks (DNN), Deep Belief Network (DBN), Recurrent Neural Network (RNN) applied to fields of computer vision, speech recognition, natural language processing etc. Deep learning as the name implies are deep structured learning architectures which use a cascade of many layers of nonlinear processing units for feature extraction and transformation. Deep learning systems are part of the broader machine learning where each layer of the deep learning systems passes a more sharpened version of the data it receives to the next layer.

3.1 Essential CNN terms

For proper understanding and to prevent obscurity in the further sections of this document, some selected CNN terms explained below.

- ***Channel:*** This is a conventional term used to refer to a certain component of an image. Coloured Images have 3 channels which are the Red Green and Blue components which make up three 2-dimensional matrices. On the other hand, a grey scaled image is made up of 1 channel (i.e. one 2-dimensional matrix) with pixel values ranging from 0 to 255 with zero (0) indicating a black and two hundred and fifty-five (255) indicating a white.
- ***Depth:*** Depth corresponds to the number of filters used for a convolution operation in CNNs
- ***Stride:*** Stride is the number of pixels by which the filter matrix would be slid over the input matrix (see section 3.2.1 below for more understanding)
- **Parameter Sharing:** Unlike Multi-Layer Perceptron (MLP) in which every neuron is fully connected to the neuron in the next layer, CNN adopts a parameter sharing mechanism whereby each neuron of the filter is connected to only a subset of the input data/image to that layer. This is based on the assumption that if one feature is useful to compute at some spatial position (x_1, y_1), then it should also be useful to compute at a different position (x_2, y_2) [27].

3.2 Typical CNN layers of operation

LeNet was one of the very first convolutional neural networks which helped propel the field of Deep Learning. This pioneering work by Yann LeCun was named LeNet5 in 1988. There have been several new architectures proposed in the recent years which are improvements over the LeNet, but they all use the main concepts from the LeNet. There are four main layers of operations in the ConvNet

1. Convolution
2. Non-Linearity (ReLU)
3. Pooling or Sub Sampling
4. Classification (Fully Connected Layer)

3.2.1 CNN convolutional layer operation

At the convolutional layer, the input collection (array, matrix) is convolved (element wise multiplication) with a filter (neuron, feature detector or kernel) which is basically also a collection of numbers called weights. The filter and the input array must share equal depths for the multiplicative function of convolution to be able to take place. The resulting array of these convolution is termed ***the feature map or activation map.*** Each position of the feature map is generated by an element wise multiplication of the kernel with the input matrix at that location and summing up the outputs of the multiplication

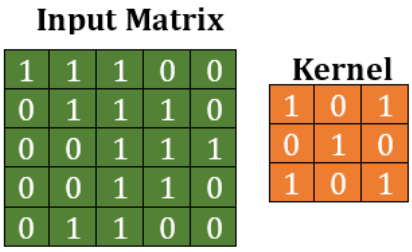

Figure 3.1 Sample Feature Input (left) And A Sample Filter Kernel (Right)

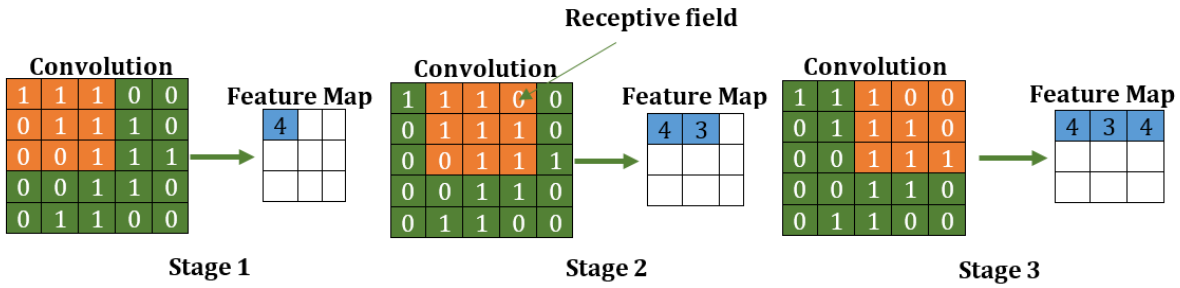

Figure 3.2 Three Stages of Convolution of the Sample Input Matrix with the Kernel to Generate the Feature Map (Stride =1)

CNNs aim to use spatial information between the pixels of an image as Convolution preserves the spatial relationship between pixels by learning image features using small squares of input data. Therefore, they are based on discrete convolution. Each of these filters at the convolution layer of the CNN can be thought of a feature identifiers, just like the biological neural cells in the cortex which only respond to certain kinds of patterns. Considering the fact that each of these filters will extract out information from the image input, the more the number of filters applied on the image input, the more (depths) different kinds of feature maps

will be generated e.g. a 4 by 3 by 72 feature map in the case where by 72 different kinds of filters are used. Other layers exist in-between the convolutional layers of the CNN, however, their primary aim is to provide nonlinearities and preservation of dimension that help to improve the robustness of the network and control overfitting. It is not restrictive that we move the kernel over the input matrix 1 pixel hop at a time during the convolution, Different stride values can be selected, however, the bigger the stride (jump steps during convolution), the smaller the feature maps that will be generated. Also, it is sometimes necessary to zero padding the input to fit a desired dimension, the size of the feature map to be used can also be regulated. In order words, even if the input matrix is too small to accommodate effectively the feature extractor during the convolution process, this can be zero padded to arbitrary widths and height to favour the feature extractor in use. Also, zero padding allows the kernel to be used to act efficiently on border values in a more efficient way.

3.2.1.1 Maintaining spatial accuracy

To make sure the strides and dimension of the filters are set appropriately, (27) below is used, as it helps calculate the number of kernels which will fit into an input volume.

$$Spatial\ Arrangement\ Validator = \frac{(W-K)+2P}{S} + 1 \qquad (27)$$

$$W = Input\ Size$$

$$K = Size\ of\ Convolution\ Layer\ Kernels$$

$$S = Stride\ Applied$$

If the resulting answer of (27) above is not an integer, then it is an indication that the strides are set incorrectly and the kernels cannot be tiled to fit across the input volume in a symmetric way. By Zero-padding the input matrix, the stride selection error can be rectified. Setting a stride of 1 ensures that the input and output volumes of the convolutional layer have the same dimension, however increases the number of computations that the network will need to carry out per time.

For example, if the result of (27) above was 55 in a layer with a depth size of 96 and input size of 11x11x3, we can easily calculate the layer's output volume to be 55x55x96. Also, due to weight sharing, i.e. each depth slice only having one/same weight value for all its neurons, the total unique weight for that layer can be computed to be 96x11x11x3 (+96 biases if each of the kernels have biases).

3.2.1.2 Channel and depth translations

For the convolution of a multiple-channel input, the filter simultaneously glides over the same receptive field across the depth of the channel, computes the dot product at the multiple channels simultaneously and then sums up the convolution result for each of the channel layers into one to produce just one channel of convolved solution. For example, an input convolution layer of depth 256 filters each of size 2x2x3 interacting with an input of size 11x11x3 with a stride of 1 will glide through the first 2 by 2 position in input 11x11x1, 11x11x2, 11x11x3, compute the convolution per layer with the filter's 2x2x1, 2x2x2, 2x2x3 respectively and then sum up the result from the three layers as the ultimate convolution result for the first 2 by 2 location of the input data. Therefore, the outcome of this convolution will result in just 256 feature maps and not 256 by 3 feature maps. In other words, a three channel input convolved with a 3 channel filter will produce a one channel output, primarily because the convolution is performed across the 3 channels simultaneously summed up to produce just one output for that activation space.

3.2.1.3 AlexNet convolution peculiarity

The convolutional blocks of the AlexNet framework (a variation of the CNN) makes use of a filter whose number of channels are half of those of the inputs data to that layer. To perform this kind of convolution, the input channels (e.g. 96) is divided into 2 groups of (e.g. 48 each) and processed independently in order to achieve the desired result. Also note that the Convolutional Layer is not only restricted to be after the input layer. Different variations of the CNN have varying numbers of Convolutional Layers placed in desired locations in the network's topology.

3.2.2 Non-Linearity (ReLU) layer operation

In most CNN architectures, after every convolution operation in the CNN, a non-linearity operation is carried out. The Rectification Linear Unit (ReLU) performs an element wise operation which replaces all negative pixel values in the feature map by zero. The purpose of ReLU is to introduce non-linearity in CNNs as real-world data are mostly non-linear while convolution is a linear operation. The output feature map of this operation is called the **'Rectified Feature Map'**. Other non-linearity function exists and can be used, however, the performance of ReLU has been noted to surpass the rest.

3.2.3 Pooling layer operation

This operation helps to reduce the dimensionality of (each of) the feature map while still retaining the key features. There are different ways in which pooling is achieved, either through the Max Pooling, Average Pooling or Sum Pooling method. In doing this, a spatial neighbourhood and stride length is defined (e.g. a 3 by 3 space, stride 1) and then the rectified feature map is looped through using this space dimension and stride. At every operation point, specific element/elements from the rectified feature map in that area is selected. If the Max Pooling technique is in use, then the maximum element from the rectified feature map of that space area is selected. In the case of Average Pooling, then the average of the elements in the space area is selected while for Sum Pooling, the sum of the elements in the area is selected.

In practice, max pooling is known to work better than the rest. The output of the Pooling operation produces equal number of maps as was inputted into it, but reduced in dimension. The function of this pooling is to progressively reduce the spatial size of the input representation thereby reducing the number of parameters and computations to be done by the network [33]. Also, helps the network retain performance, even in the event of minor changes in input image, thereby providing an equivariant representation of the image.

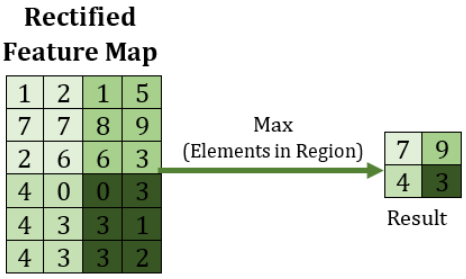

Figure 3.3 Max Pooling with a 2 by 2 filter and a stride of 2

3.2.4 The fully connected layer operation

The Fully Connected layer is a traditional Multi-Layer Perceptron that uses a SoftMax activation function in the output layer. The term "Fully Connected" implies that every neuron in the previous layer is connected to every neuron on the next layer. The output from the

convolutional and pooling layers represent high-level features of the input image while the purpose of the Fully Connected layer is to use these features for classifying the input image into various classes based on the training dataset. Most of the features from convolutional and pooling layers may be good for the classification task, but combinations of those features might be even better which explains the benefit of adding the fully connected layer. The sum of output probabilities from the Fully Connected Layer is 1. This is ensured by using the SoftMax as the activation function in the output layer of the Fully Connected Layer. The SoftMax function takes a vector of arbitrary real-valued scores and squashes it to a vector of values between zero and one that sum to one. The SoftMax function helps to get the probability distribution of each output from a group of outputs. The SoftMax classifier is hence minimizing the cross-entropy between the estimated class probabilities.

As has been said, it is a generalization of the logistic function that "squashes" a K-dimensional vector z of arbitrary real values to a K-dimensional vector σ(z) of real values in the range (0, 1) that add up to 1. The function is given by

$$\sigma(z)_j = \frac{e^{z_j}}{\sum_{K=1}^{K} e^{z_k}} \quad (28)$$

3.3 Common CNN architectures

Variations in the CNN is achieved today by reconfiguring the arrangement of the layers it is comprised of [28]. In designing the CNN Architecture to use for an operation, a lot of intuition goes into it. However, some guidelines exist in helping with the choice of the number of filters to use [29].

3.4 Training CNNs

The CNN employs a similar training technique as those used by the traditional MLP (ANN) discussed in Section 2.2.3 of this report. In summary, this technique involves the implementation of a Backpropagation mechanism which further reduces the degree of error contributed to the final output by each parameter/weight in the network. An iterative process of doing this fine tunes the network weights to generate the right classification probabilities for the inputs passed through it. Initially in the training process, random weight values are applied to the kernels and the neurons of the Fully Connected Layers.

3.4.1 Flow of operation of the network (training stage)

Steps 1 to 7 below explain the iterative process undergone by a typical CNN before it converges into a fully learned system capable of making near accurate predictions.

- ***Step 1:*** Initialize all filters and Fully Connected Layer (FCL) neuron weights with random values
- ***Step 2:*** Input training image. Image goes through the forward propagation steps (convolution, ReLU and pooling etc., depending on the specific CNN architecture)
- ***Step 3***: Probabilities of the output of the final Fully Connected Layer is found.
- ***Step 4:*** The output probability is compared with the targeted probability
- ***Step 5:*** Total error of the system is calculated
- ***Step 6:*** Backpropagation is used to calculate the gradients of the error with respect to all weights in the network and gradient descent is used to update all filter and neuron values in a manner such as to minimize the output error.
- ***Step 7:*** Step 2 to 6 are repeated continuously, until there is very minimal error between the output probability of the network and the targeted output probability.

Training the ANN requires a conscious analysis, to ensure that the network is not a victim of over-fitting (over training) and when finally, the system has been correctly trained,

the weights can be "frozen". At times, the system is designed not to lock itself in but rather continue to learn while in production use. This is entirely dependent on the decision of the system architect. In some industry application cases, the finalized network is not only frozen, but also turned into hardware so that its operation can be fast.

3.5 AlexNet and the evolution of CNN

LeNet was developed sometimes around the 1990s, however, between the 1900s and 2012, there existed only little improvements in the field of CNN (incubation period). In 2012, AlexNet, a variation of the traditional (LeNet) CNN was developed by Alex Krizhevsky. It was a deeper and much wider version of the LeNet which became a significant breakthrough with respect to the previous approaches. The current widespread application of CNNs can be attributed to this work. Between 2012 and 2017, other improved CNN adaptations have been released such as ZF Net (2013), GoogLeNet (2014), VGGNet (2014), ResNets (2015) and DenseNet by Gao Huang in 2016. ***The AlexNet model is implemented in this project*** both on CPU and on an FPGA, for real-time traffic sign recognition and detection.

The AlexNet, CNN variation developed in 2012 by Alex Krizhevsky, Ilya Sutskever and Geoff Hinton consists mainly of eight layers. First 5 of the layers are convolutional and the rest are fully connected layers. The first and second convolutional layers are followed by response-normalization layers, then these response-normalization layers are followed by max pooling layers. In addition, the fifth convolutional layer is also followed by a Max pooling layer. The output of every convolutional layer and fully connected layer is put through ReLU non-linearity while the output of the last fully connected layer is sent to the 1000-way SoftMax layer which produces 1000 probability values for 1000 class labels, where higher value corresponds to higher probability.

For this project, only 43 classifications are necessary, therefore, the SoftMax function is queried and applied across feature channels and in a convolutional manner at all spatial locations [30] to produce only 43 probabilities. Recent applications of the AlexNet model on the ImageNet Large Scale Visual Recognition Challenge (2012) yielded a top-5-error rate as low as about 19.7% (measurement of the number of times whereby the target result were not part of the network's top 5 predictions). This challenge tasked developers to design and implement an Artificial Intelligent (AI) network to classify over a million images into multiple (about 1000) categories.

Table 3.0 below shows a tabular representation of the AlexNet model and its parameters while Figure 3.4 shows it in its diagrammatic form

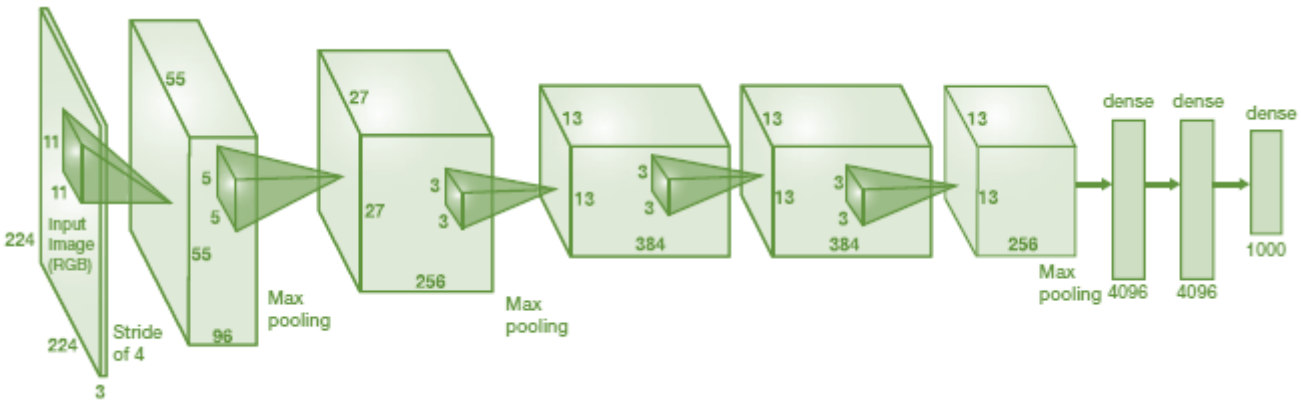

Figure 3.4 Figure Showing the AlexNet Topology

Table 3.0 Tabular Representation of the AlexNet Model

CONVOLUTIONAL BLOCK 1							
LAYER NAME	**LAYER TYPE**	**INPUT SIZE**	**OUTPUT SIZE**	**FILTER SIZE**	**FILTER DEPTH**	**STRIDE**	**PADDING**
Input	Input	227x227x3	227x227x3	null	null	null	null
Conv1	**Convolutional**	**227x227x3**	**55x55x96**	**11x11x3**	**96**	**4**	**0**
Relu1	Response Linearity	55x55x96	55x55x96	null	null	null	null
Lrn1	**Response Normalization**	**55x55x96**	**55x55x96**	**null**	**null**	**null**	**null**
Maxpool1	Max Pooling	55x55x96	27x27x96	3x3	96	2	0
CONVOLUTIONAL BLOCK 2							
LAYER NAME	**LAYER TYPE**	**INPUT SIZE**	**OUTPUT SIZE**	**FILTER SIZE**	**FILTER DEPTH**	**STRIDE**	**PADDING**
Conv2	**Convolutional**	**27x27x96**	**27x27x256**	**5x5x48**	**256**	**1**	**2**
Relu2	Response Linearity	27x27x256	27x27x256	null	null	null	null
Norm2	**Normalization**	**27x27x256**	**27x27x256**	**null**		**null**	**null**
Maxpool2	Max Pooling	27x27x256	13x13x256	3x3	256	2	0
CONVOLUTIONAL BLOCK 3							
LAYER NAME	**LAYER TYPE**	**INPUT SIZE**	**OUTPUT SIZE**	**FILTER SIZE**	**FILTER DEPTH**	**STRIDE**	**PADDING**
Conv3	**Convolutional**	**13x13x256**	**13x13x384**	**3x3x256**	**384**	**1**	**1**
Relu3	Response Linearity	13x13x384	13x13x384	null	null	null	null
CONVOLUTIONAL BLOCK 4							
LAYER NAME	**LAYER TYPE**	**INPUT SIZE**	**OUTPUT SIZE**	**FILTER SIZE**	**FILTER DEPTH**	**STRIDE**	**PADDING**
Conv4	**Convolutional**	**13x13x384**	**13x13x384**	**3x3x192**	**384**	**1**	**1**
Relu4	Response Linearity	13x13x384	13x13x384	null	null	null	null
CONVOLUTIONAL BLOCK 5							
	LAYER TYPE	**INPUT SIZE**	**OUTPUT SIZE**	**FILTER SIZE**	**FILTER DEPTH**	**STRIDE**	**PADDING**
Conv5	**Convolutional**	**13x13x384**	**13x13x256**	**3x3x192**	**256**	**1**	**1**
Maxpool3	Max Pooling	13x13x256	6x6x256	3x3	256	2	0
Relu5	**Response Linearity**	**6x6x256**	**6x6x256**	**null**	**null**	**null**	**null**
FULLY CONNECTED LAYER 1							
LAYER NAME	**LAYER TYPE**	**INPUT SIZE**	**OUTPUT SIZE**	**FILTER SIZE**	**FILTER DEPTH**	**STRIDE**	**PADDING**
Fc1_Conv6	Converted to Convolutional	6x6x256	1x1x4096	6x6x256	4096	1	0
Relu6	**Response Linearity**	**1x1x4096**	**1x1x4096**	**null**	**null**	**null**	**null**
FULLY CONNECTED LAYER 2							
LAYER NAME	**LAYER TYPE**	**INPUT SIZE**	**OUTPUT SIZE**	**FILTER SIZE**	**FILTER DEPTH**	**STRIDE**	**PADDING**
Fc2_Conv7	Converted to Convolutional	1x1x4096	1x1x4096	1x1x4096	4096	1	0
Relu7	**Response Linearity**	**1x1x4096**	**1x1x4096**	**null**	**null**	**null**	**null**
FULLY CONNECTED LAYER 3							
LAYER NAME	**LAYER TYPE**	**INPUT SIZE**	**OUTPUT SIZE**	**FILTER SIZE**	**FILTER DEPTH**	**STRIDE**	**PADDING**
Fc3_Conv8	Converted to Convolutional	1x1x4096	1x1x43	1x1x4096	43	1	0
Relu8	**Response Linearity**	**1x1x43**	**1x1x43**	**null**	**null**	**null**	**null**
SOFTMAXLOSS LAYER (Outputs the Prediction Probabilities)							

The sections 3.5.1 and 3.5.2 of this gives further explanations into the AlexNet model and its peculiarities.

3.5.1 AlexNet peculiarity (Response Normalization layer)

AlexNet incorporates an additional feature/layer into its architecture which is called the Response Normalization Layer (LRN). This mimics the neurobiological concept of 'lateral inhibition' whereby an excited neuron has the capacity to subdue its neighbours [31]. This is very essential, mainly because ReLU neurons have unbounded activations, therefore requiring the LRN normalize them. There are two major approaches to carrying out the LRN. Either within the same layer or across layers. Both these methods tend to amplify the excited neuron while dampening the surrounding neurons.

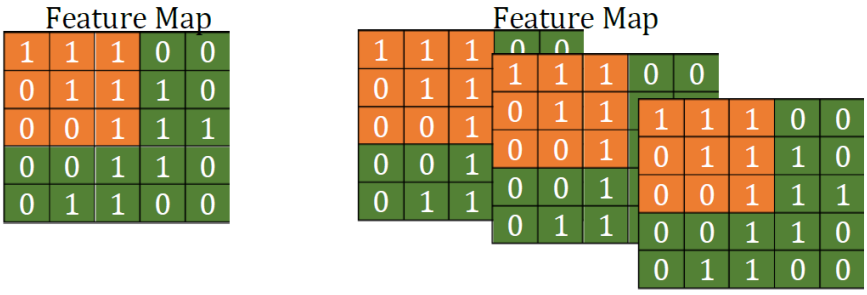

Figure 3.5 LRN Within the Same Channel (Left) and LRN Across Channels (Right)

For LRN within layers, only a 2-D neighbourhood of N-by-N is considered, however, for the method across channels, corresponding N-by-N neighbourhoods in the other layers are also considered. E.g., in Figure 22 (Right) above, a 3-by-3-by-3 space will be used in computing the normalization for the activation area.

The normalized response of the activity ${b^i}_{x,y}$ of applying Kernel 'i' at position (x, y) of the input matrix followed by the ReLu non-linearity ${a^i}_{x,y}$ is given by the (29) below

$$ {b^i}_{x,y} = \frac{{a^i}_{x,y}}{\left(k+\alpha \sum_{j=\max(0,i-\frac{n}{2})}^{\min(N-1,i+\frac{n}{2})} ({a^j}_{x,y})^2\right)^{\beta}} \tag{29} $$

For this project, just as we have in the AlexNet model, the sum runs over the (N) adjacent feature maps at the same spatial position where N is the total number of kernels in the layer (depth). The constants k, n, α, and β of (29) above are hyper-parameters whose values were determined by the AlexNet developers through the use of a validation set. The constants derived from the validation sets are as listed below

$$ k = 2 $$

$$ n = 5 $$

$$ \alpha = 10E-4 $$

$$ \beta = 0.75 \text{ [32]}. $$

3.5.2 Converting the fully connected layers to convolutional layers

The noticeable difference between the fully connected layer and the convolutional layer of the CNN is the fully connected nature of the neurons in the FCL i.e., unlike the convolutional layer neurons which are connected only to specific local regions of the input, the neurons of the fully connected layer are all connected to each other. This notwithstanding, it is important to note that the mode of operation of the neurons in both layers remain similar. It is therefore feasible to convert the fully connected layer into a convolutional layer. This can be

achieved by evaluating the input values to be collected by the FCL and introducing a convolution filter, stride and padding value with dimension which will result in a spatial arrangement of 1 using (27) above. For example, a 4096 neuron FC layer with an input of 6x6x256 can be converted into a convolution layer by using 4096 convolution filters of size 6x6x256 with 0 padding and a stride of 1. Plugging in this values into (27) will realize a spatial arrangement of 1 as shown below

$$\frac{(6-6)+2(0)}{1}+1=1$$

In order words, the result of this convolution layer will be a 4096 1x1 feature maps. When paired up with a next convolution layer with also 4096 1x1 filter this will be equivalent to having 2 layers of fully connected 4096 neurons.

4.0 Design, Implementation, Training and Testing of The Vision System's ConvNet Model

The MatConvNet Toolbox is used to create, modify and train the AlexNet CNN model theoretically explained in the chapter 3 of this project report, as it is the designated CNN model to be implemented for the vision system. MatConvNet is a MATLAB toolbox used for implementing Convolutional Neural Networks (CNN) for computer vision applications [33]. It is open-source released under a Berkeley Software Distribution (BSD)-like license (a family of permissive free software licenses, imposing minimal restrictions on its redistribution). MatConvNet can be used to replicate the architecture of many of the well know CNNs, however, its development team has also made available pre-trained models of these architectures. Notwithstanding, this project develops from the scratch its own AlexNet model, trains and tests its performance level using a brand new IMDB fabricated from the 30, 209 traffic sign images made available on the German Traffic Sign Recognition Benchmark (GTSRB) website ([34]).

The objective of this chapter is to show the methodology employed in:

1. Designing the selected CNN model (AlexNet) for robust traffic sign classification in the proposed vison system using MATLAB
2. Creating an image database (IMDB) which can be used to train and test the designed CNN architecture.
3. Training, testing and evaluating the performance of the implemented CNN network

MatConvNet includes a variety of layer models contained in its MATLAB library directory, such as convolution, deconvolution, max and average pooling, ReLU activation, sigmoid activation and many other pre-written functions. There are enough elements available to help implement many interesting state-of-the-art networks out of the box, or even import them from other toolboxes such as Caffe.

After the creation of the AlexNet model, the network undergoes training and is then tested to ensure performance and accuracy. Considering the fact that two levels of operation is cascaded, one for the traffic sign (object) detection and the other for the actual classification of the traffic sign, it is important to note that this chapter only explains the development, training and testing of the image classifier. Detailed testing and verification is carried out to ensure optimal performance of the system.

In order to develop a Convolutional Neural Network which is able to classify images fed into it, the network has to be trained over multiple epochs in a specialized manner. Batches of training images fed into the CNN first have to be pre-processed to the network's standard input size and in most cases, normalized to have zero mean. This initiative affect the rate of convergence of the network during training to a great extent. It is important to remember that the convolutional layers of the network serve as the feature extractors while the fully connected layers and the softmax serve as the processing and classifier elements

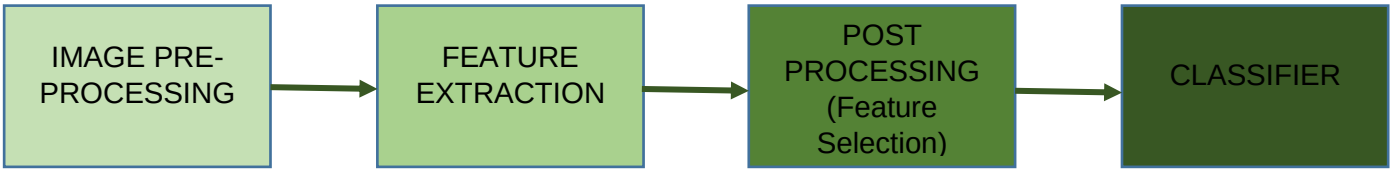

Figure 4.0 Block Diagram Showing Data Flow Through the CNN During Training

4.1 CNN network structure (MATLAB MatConvNet)

There are two primary wrappers that can be used to create a CNN using MatConvNet in MATLAB, and these are the SimpleNN and the DagNN. The sequential network structure (SimpleNN) in MATLAB/MatConvNet library purely requires that given an input x into a network, evaluating the network is simply a matter of evaluating all the blocks the network is composed of from left to right. However, other types of networks structures exist which define that a layer is ready for evaluation as soon as all its required inputs have been evaluated and are ready to act. This is called the Direct Acyclic Graph (DAG). Direct Acyclic Graph Neural Networks (DagNN) wrapper are object-oriented wrapper for CNN with complex topologies. Here, functions can have any number of inputs or outputs. Also, variables and parameters can have an arbitrary number of outputs.

For this project, both the SimpleNN and DagNN structures are tested out however, the SimpleNN is finally utilized in order to avoid further complexity. An excerpt of the codes for the creation and training of the SimpleNN version of the AlexNet network is shared in Appendix B.2 of this document.

4.2 CNN training dataset creation

A dataset of traffic sign images from the German Traffic Sign Recognition Benchmark (GTSRB) website is compiled and used to create the image database (IMDB) used to train and test the deep neural network. The German Traffic Sign Benchmark is a multi-class, single-image classification challenge held at the International Joint Conference on Neural Networks (IJCNN) 2011. The dataset consists of over 39,000 images in total, grouped into 43 different road traffic symbols.

Through a written IMDB creation script, the dataset is split into 70% training, 20% validation and 10% test sets of images which are used in the holistic training, validation and testing of the created AlexNet model. Validation images are used to test the performance of the network during the training process while the test images are reserved in this project to perform personal test on the network, post training. The validation set actually can be regarded as a part of training set, but it is usually used for parameter selection and to avoid overfitting. If a model is trained on a training set only, it is very likely to get close to 100% accuracy and over fit, thus get very poor performance on test set which have never been seen by the network before. The test-sets are only used to test the performance of a trained model and are the best means of detecting over fitting in the network. In summary:

- ***Training set:*** Used to fit the parameters (weights)
- ***Validation set:*** Used to tune the parameters (architecture)
- ***Test set:*** Used to assess the generalization and predictive power of the network (performance)

Table 4.0 below shows a list of all 43 classes of signs which the R-CNN s trained to classify, their visual and textual description. Textual descriptions/definition for the various traffic signs are gotten from [35].

Table 4.0 Table Showing the 43 Classes of Road Traffic Sign Which the CNN to Classify

S/N	TRAFFIC SIGN DESCRIPTION/DEFINITION	SIGN
1	Speed 20	
2	Speed 30	
3	Speed 50	
4	Speed 60	
5	Speed 70	
6	Speed 80	
7	End of Speed 80	
8	Speed 100	
9	Speed 120	
10	No Car Overtaking	
11	No Truck Overtaking	
12	Priority Road	
13	Priority Road 2	
14	Yield right of Way	
15	Stop	
16	Road Closed	
17	Maximum Weight Allowed	

18	Entry Prohibited	
19	Danger	
20	Curve Left	
21	Curve Right	
22	Double Curve Right	
23	Rough Road	
24	Slippery Road	
25	Road Narrows Right	
26	Work in Progress	
27	Traffic Light Ahead	
28	Pedestrian Crosswalk	
29	Children Area	
30	Bicycle Crossing	
31	Beware of Ice	
32	Wild Animal Crossing	
33	End of Restriction	
34	Must Turn Right	
35	End of No Truck Overtaking	
36	Must Turn Left	

37	Must Go Straight		
38	Must Go Straight or Right		
39	Must Go Straight or Left		
40	Mandatory Direction Bypass Obstacle		
41	Mandatory Direction Bypass Obstacle 2		
42	Traffic Circle		
43	End of No Car Overtaking		

Code B.4 of the Appendix B.3 shows an excerpt of the MATLAB script written to create the image database used for the training of the ConvNet

4.3 CNN training process and special considerations

Special considerations are given in key areas while on the training process to prevent skewed performances from the network, mainly as a result of over fitting. Overfitting happens when the network becomes too familiar with only the training data, hence loses the ability to generalize new data fed into it. While the network's error rate may continue to reduce throughout the training process, it is important to note that over fitting will begin to set in at some point in time is training continues over several more epochs.

Another important consideration made during the creation of the CNN was the method of initialization of the network's parameters. A carefree initialization of the network weights may result in the network not being able to converge during the training process. To tackle this effect, various mechanisms are employed. Figure 4.2 below shows the improvement in performance of the system (reduction in generalization error) and fast convergence rate as the training of the network progressed from Epoch 1 to 58.

4.3.1 Important terms

- ***Top1Error:*** The fraction of times in which the proposed label by the network is not the right label
- ***Top5Error:*** The fraction of test images for which the correct label is not among the top 5 most probable labels suggested by the network
- ***Batch Normalization:*** A technique used in neural networks whereby each layer of the network is provided with inputs which have zero mean per unit variance. It helps to solve the vanishing gradient descent challenge and at the same time increases the network's learning rate

4.3.2 Reducing overfitting

Reducing overfitting is achieved by transforming the images in the dataset to produce as many possible variations of them as possible. Transformations could include mirroring, blurring, reflecting, brightening and so on. These data augmentation strategies used in

reducing overfitting are broadly classified into those which have to do with generating image translations, horizontal reflections and those that are majorly about altering the intensities of the RGB channels in training images. Images in all the 43 classes of the dataset made use of from the GTSRB have been well augmented using the two main techniques highlighted here.

4.3.3 Initializing the network

If all the weights of the network are initialized to the same value, it will be difficult to break the symmetry of the network. There are therefore various proposed methods of CNN initialization, few of which are the:

- Gaussian distributed initialization method
- He's method of network initialization
- Xavier's Method

Weight initialization influence a neural network's ability to learn. It is critical for training a neural network. Many ways for initialization have been proposed based on different ideas, however, the idea from both Xavier and He's initialization is to preserve variance of activation values between layers as variance larger or smaller than one may cause activation outputs to explode or vanish [36]. Xavier initialization works well with linear activation functions while the He initialization takes the ReLU activation into account. By using the He's method instead of Gaussian random values to initialize the weights of the network, the classification and validation errors will decrease more sharply. For this project, the He's weight initialization method is used

4.3.4 Batch concept and normalization

Performing gradient descent simultaneously on the entire dataset all at once during every epoch of the training will be a very slow operation and will take too long a time to converge. This kind of method is not suited for deep architectures, therefore, during the training process of the network, batches of images are processed per time and the gradient tuning is performed after every batch (mini-batch stochastic gradient descent). In essence, instead of computing exact gradients for the updates on all the dataset, estimates are made based on the mini-batch (e.g., 100 images per batch).

The concept of batch normalization which has to do with normalizing these batches of images passed between the layers of the network helps relax the effect that may come from not properly initializing weights and learning parameters of the network. During back propagation, the gradients have to compensate for outliers which may therefore lead to many more epochs before convergence. Batch normalization regularizes the gradient from the distractions of the outliers by normalizing within the range of the mini batch. This project does not use batch normalization, but rather pre-processes the IMDB images by subtracting the mean of the entire images in the IMDB from each of the IMDB images (mean image subtraction).

4.3.5 Momentum value choice

During training in neural networks, the update direction tends to resist change, and sometimes, the network may find it difficult to escape from a local minimum, thus momentum in neural networks helps the network get out of these local minima points so that a more important global minimum can be discovered. It is important to note that too much of momentum may create issues as well.

In a nutshell, momentum simply adds a fraction 'm' of the previous weight update to the current one during the back propagation period. When the gradient keeps pointing in the same direction, this will increase the size of the steps taken towards the minimum. It is therefore often necessary to reduce the global learning rate μ when using a lot of momentum

(m close to 1). If a high learning rate is combined with a lot of momentum, the minimum will most likely be missed with huge steps [37]. This project makes use of a momentum of 0.9 and a global learning rate of 0.00005.

4.3.6 Weight decay

When training neural networks, it is common to use "weight decay," where after each update, the weights are multiplied by a factor slightly less than 1. This prevents the weights from growing too large, and can be seen as gradient descent on a quadratic regularization term [38].

4.4 Testing the developed CNN

To test the performance of the network, test scripts were written to perform different kinds of automated test on the trained network/system. During the training phase of the system, the network structure is saved after every epoch and by so doing, the improvement in performance of the network per epoch can be tested manually after each epoch.

4.4.1 Single network test

A test script was written which works with the IMDB and a trained network. The script uses the test images (labelled 'Set 3') in the IMDB to test the performance of any trained network parsed to it. These test images have never been seen by the network before, making them the best set of images to help detect overfitting. That is, to help know if the networks error rate is only low on the test images which the network may already have gotten used to while losing its ability to generalization previously unseen images. A total of 3920 test images are used as the test script is designed to be able to carry out automated test on the network as well as manual one by one testing depending on the preference of the user. The results of the test operations are shown in the Figure 4.1 to 4.5 of this report.

4.4.2 Multi-epoch test

Since the network structure is saved after each training epoch, all 58 states of the network are tested to see the improvement in network performance as the training progressed. Testing this networks individually has its benefit in the sense that the last sets of network which show very low error rates might be victims of overfitting already. By testing all post epoch networks with never previously seen test images, and plotting their performance graph, the stage at which overfitting sets in can be easily detected. Figure 4.2 below shows the performance evaluation bar graph for the first 20 different networks derived one after the other during the 58-epoch training process. The result shows that the network has not fallen prey to overfitting yet and can still go for a couple more rounds of training to provide even better results. If the network was only merely used to the training data, noticeable decrease in network performance should be seen in the evaluation bar graph shown in Figure 4.2 below. On the contrary, network's performance is shown to improve almost after every epoch till the 9^{th} epoch where performance improvement becomes almost unnoticeable up until epoch 20 where training is finally stopped.

Code B.2 in Appendix B.4 shows an excerpt of the 'Multi Epoch Analyser' script written to test all the output network produced after each epoch of the training process

Figure 4.1 to 4.6 below show the results of the performance analysis and testing carried. In total, 58 epochs (rounds) of training was carried out in *47 hours using over 39000 training images* on a 16 Gigabyte RAM quad core processor. Figure 4.1 below shows the descent in error rate of the classifier/CNN over the course of the 58 epochs. See section 4.3.1 of this report for definitions of the top1 and top5 error rate.

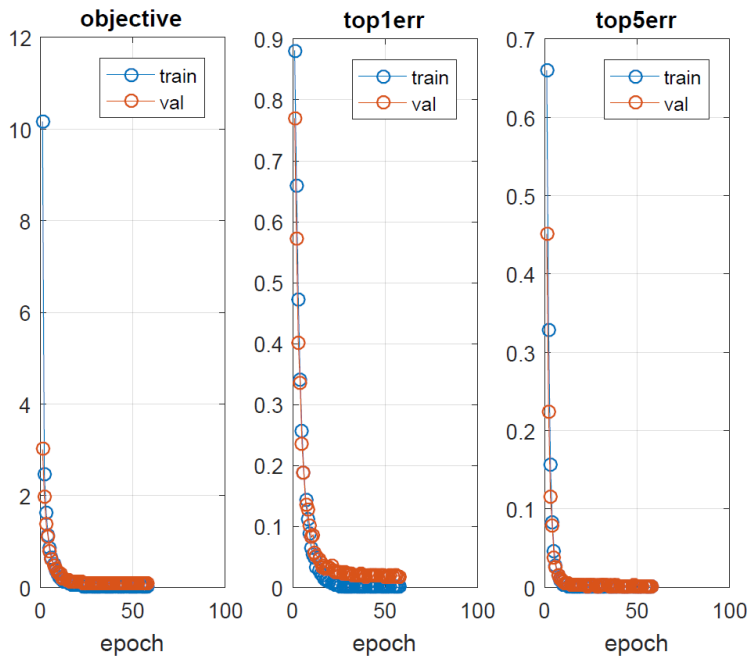

Figure 4.1 Graph Showing Error Descent as Training Progressed Between Epoch 1 to 58

The AlexNet network structure is saved after each epoch and testing is carried out on each of the saved network to ascertain the best performing network and also detect overfitting (see section 4.3.2 of this report for more explanation on overfitting).

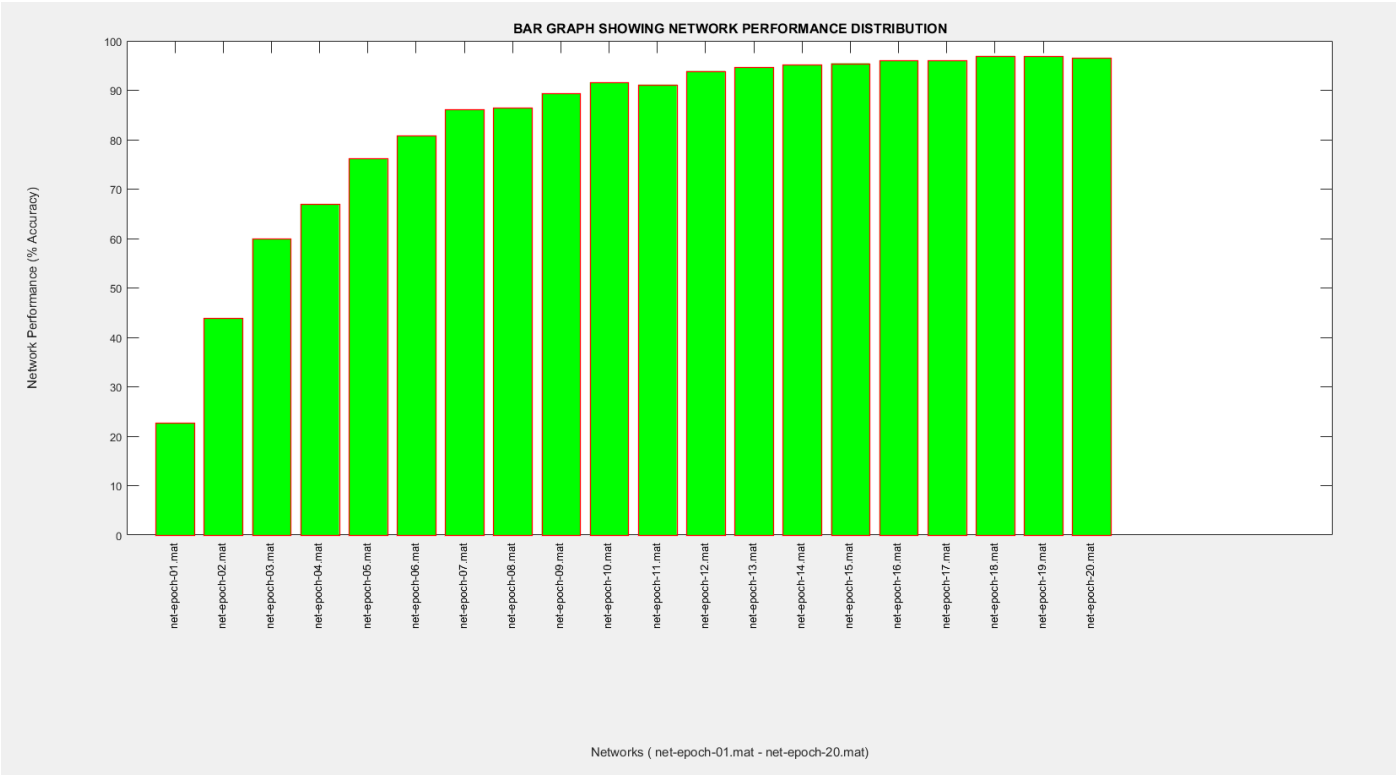

Figure 4.2 Multi-Epoch Performance Test Result

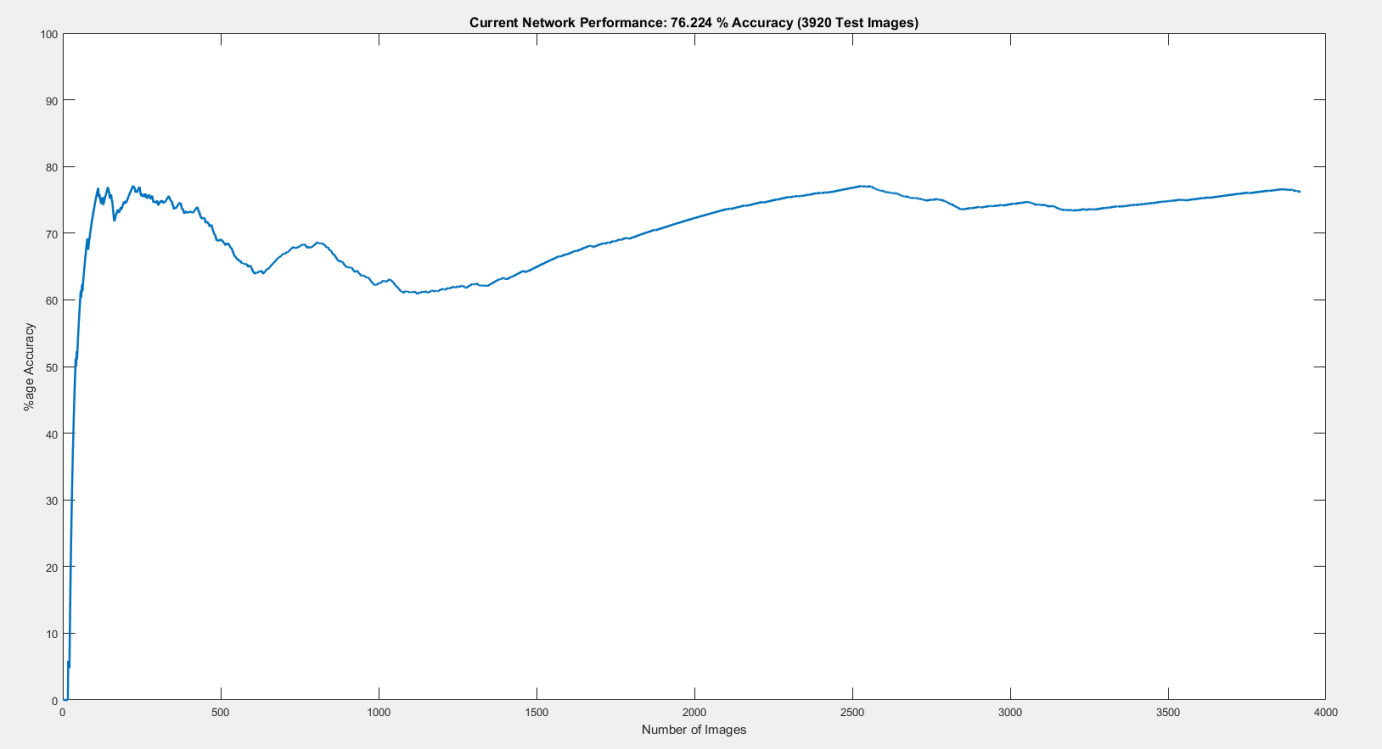

Figure 4.3 Network's Performance Level After the 5th Training Epoch

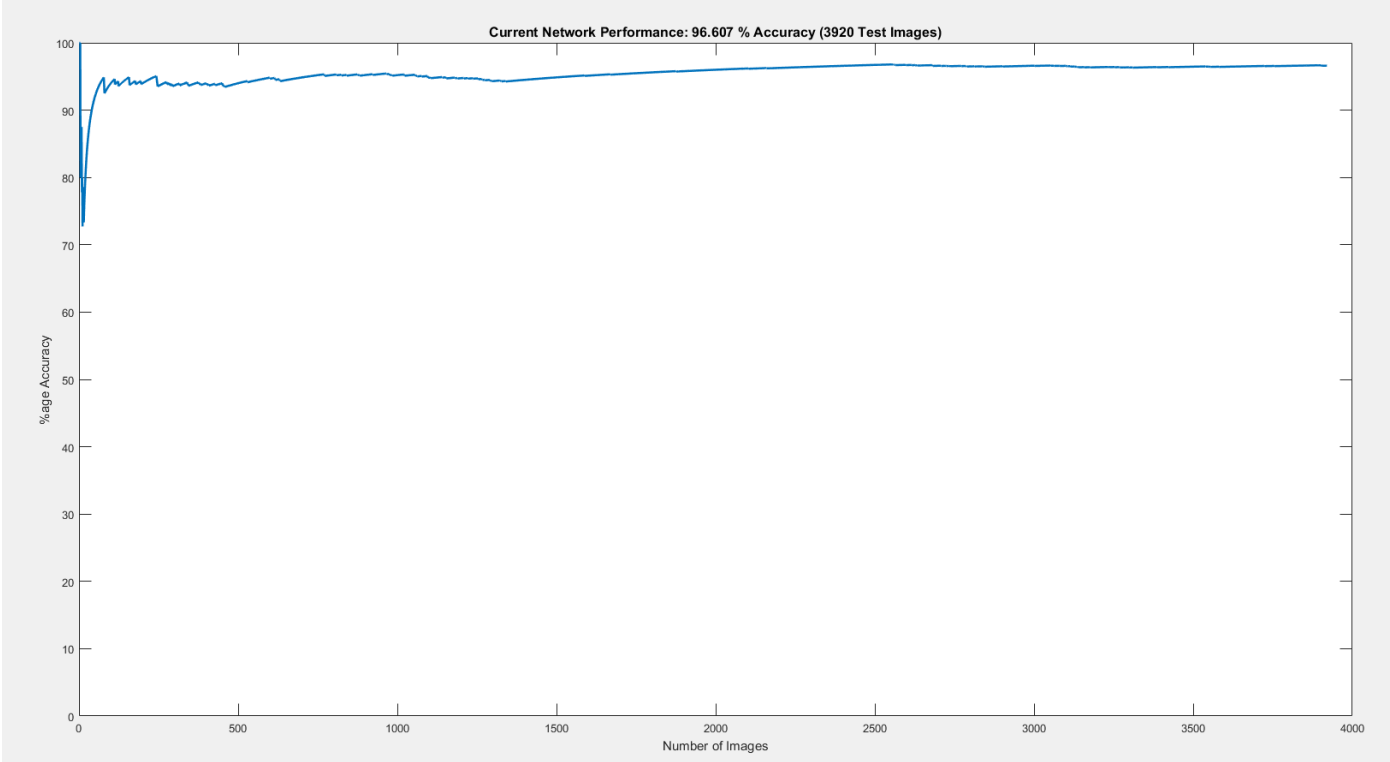

Figure 4.4 Network's Performance Level After the 20th Training Epoch

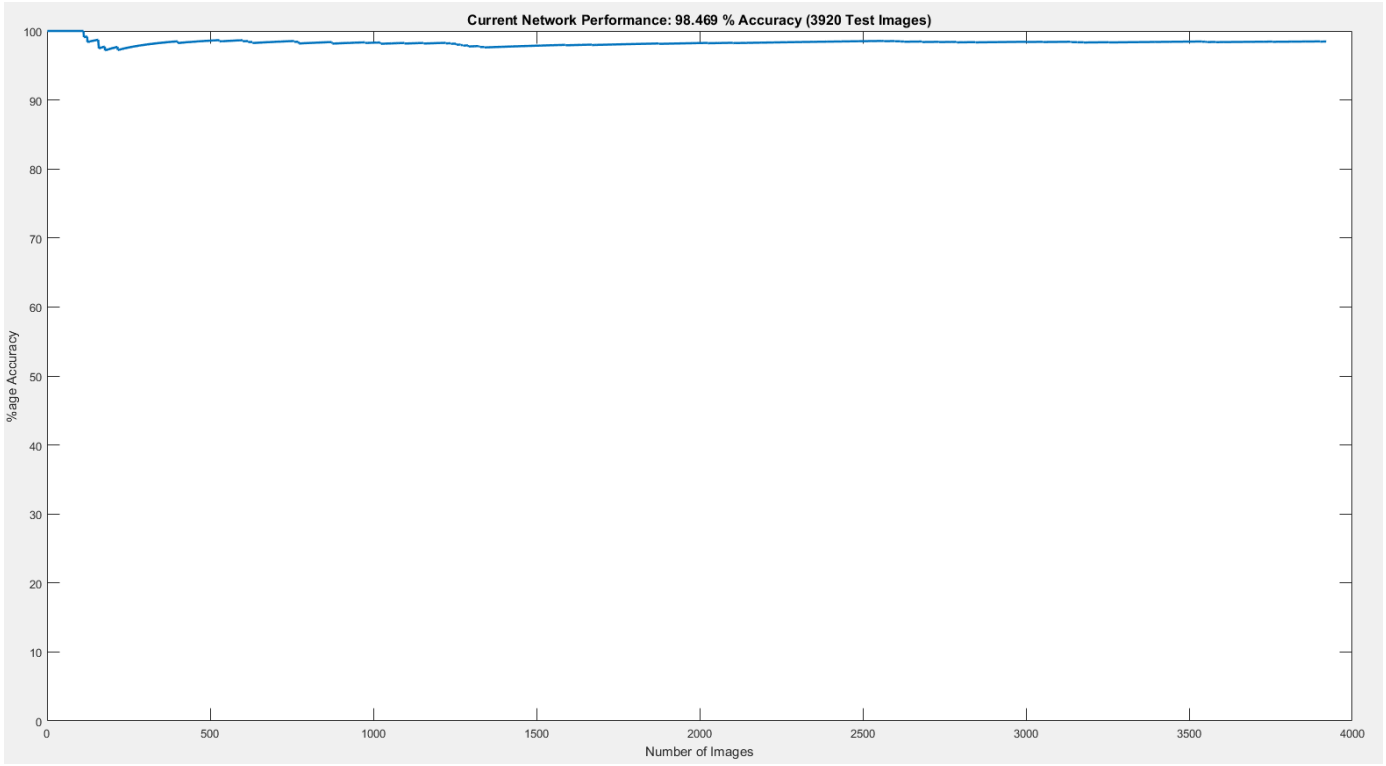

Figure 4.5 Network's Performance Level After the 58th Training Epoch

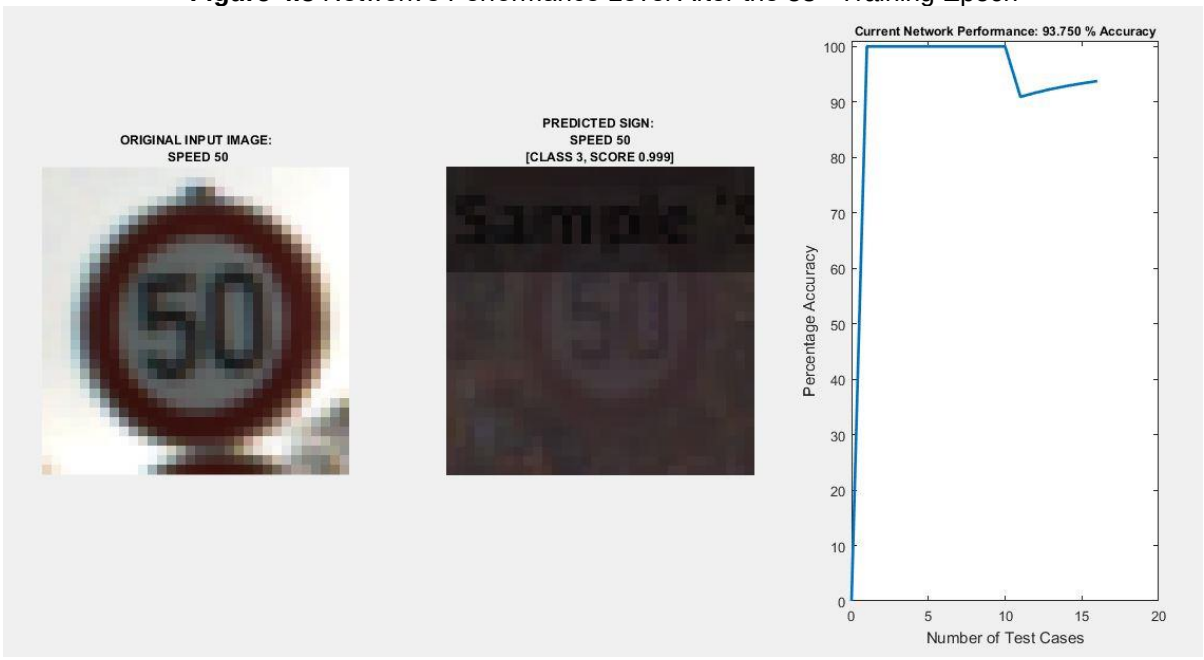

Figure 4.6 Performing One-By-One Testing on the 58th Epoch Network

Figure 4.7 below shows the result of a similar random test performed on the 58th epoch trained network which yielded a correct result.

Figure 4.7 Result of a Random Tests Carried Out on the Trained Network (58th Epoch)

5.0 Development of The Vision System's Sign Detection Mechanism

Detecting and classifying objects of various sizes in a scene is an important sub-task in Computer Vision, as most of these objects of interest only occupy a small fraction of the entire image under analysis. Quite a number of the previous CNN image processing solutions target objects that occupy a large proportion of the image, and as we know, such solutions do not work well for target objects occupying only a small fraction of the image area [39]. A typical example of these occurrences are in traffic scenes whereby the traffic light of interest only resides in a small fraction of the actual image. It is therefore important to device solid scene classification and object detection methodologies which will perform well even when the ROI is only a small but significant part of the entire image.

The fact that traffic signs are categorized according to function and are majorly in common shapes of circle, triangle, square and octagons while maintaining a combination of white, yellow, red and blue colours, makes it intuitively deductible that traffic scene classification can be optimally carried out by a combination of colour and shape analysis. The detection system uses a combination of information gleaned from this separate analysis to predict a Region of Interest (ROI). The sign classification operation follows this step to determine which specific kind of sign is present (if any). It is important to keep in mind that while detection means finding the Region of Interest, the actual classification is the one which analyses the detected region so as to provide a label to it.

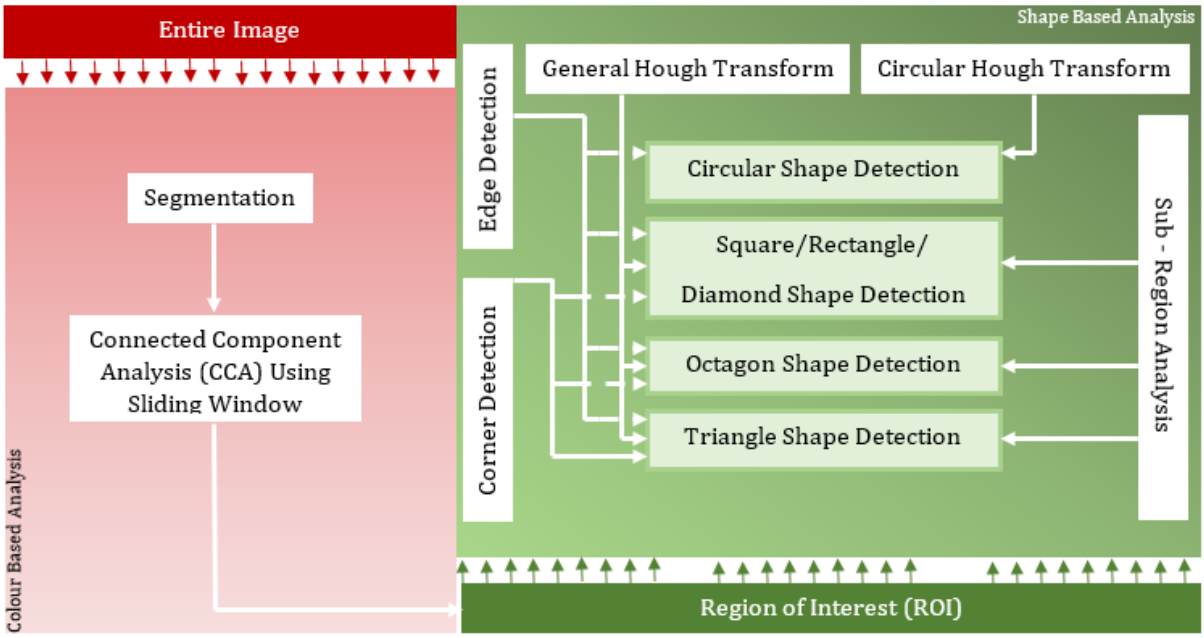

Figure 5.0 Block Diagram Showing the Processes Involved in the Sign Detection

The main objective of this chapter is to elucidate the development and testing process for the vision system's traffic sign detection module which will work hand in hand with the Neural Network to ensure accurate detection of traffic signs within a traffic scene before attempt is made to classify such regions of interests.

5.1 Initial ROI proposal based on colour analysis

Due to the distinct colour characteristics shared among traffic signs, the first step in acquiring a region proposal in this project is done through colour analysis of the input image i.e. colour based segmentation of regions of interest suspected to be road signs [40]. In this method, images passed to the system are thresholded at particular intervals, thereby leaving behind only regions containing colours of interest. However, challenges usually arise due to the variations in colour as a result of change in weather condition, illumination, and fading due to exposure to sun-light. For this reason, the colour based detection for this Vision System is conducted in the Hue, Saturation, Value (HSV) colour space i.e. each frame before being processed is converted to the HSV colour space. For several other projects, this has been attempted within other colour spaces such as Hue Saturation Intensity (HIS), Commission Internationale de l'Eclairage Lab (CIElab), Luminance, Blue Chrominance, Red Chrominance (YCbCr), Red, Green and Blue (RGB) and Cyan, Magenta, Yellow, and Key (CYMK) [40]. For clarity, note that the colour space simply refers to the range of colours, or gamut, that a camera/device/eye etc can see. The sections below gives more explanation of the RGB and HSV colour spaces.

5.1.1 RGB Colour Space

RGB (Red, Green, and Blue) colour space is the basic colour space widely used for computer graphics. It is built in the form of a cube in the Cartesian coordinate system in which the x, y and z axes are represented by R, G and B respectively. Its major drawback is around the complexity that arises between RGB colours and external illumination effects which most times leads to an apparent change in the actual colour, with clusters of colours been shifted towards the black or white colour corners of the colour spaces. It therefore means that segmenting an object using RGB space with the same set of thresholds under varying light conditions such as night and day will be inaccurate.

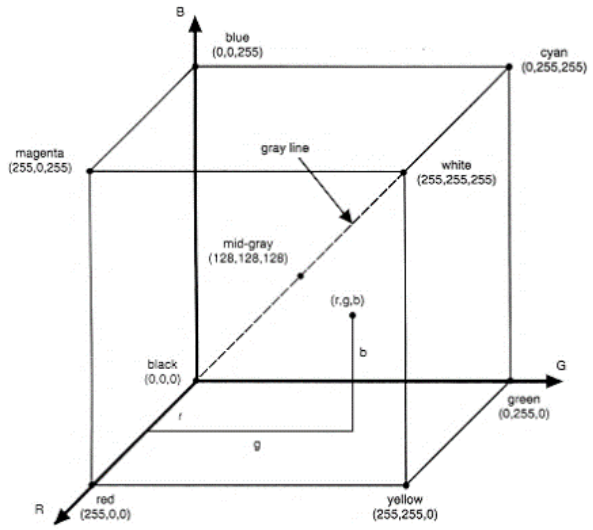

Figure 5.1. RGB Colour Space [41]

5.1.2 HSV Colour Space

HSV colour space on the other hand is designed to be very close to the way the human eye perceives colours. In its circular gamut, the Hue component contains the original colour itself, while Saturation contains information on number of pixels available for a colour. The Value component on the other hand contains information related to brightness or darkness of a pixel. The Hue component represents the colours themselves and different shades of these colours can be obtained by moving 0 to 360 degrees within the Hue space. The saturation

level of these colours increases towards the edge of the circle and it decreases when it tends towards the centre. Value (brightness) increases as it moves towards white colours and decreases towards black colours.

Although different software application use different ranges to represent minimum and maximum HSV values, OpenCV uses the range [0,179], [0,255], [0,255] for its HSV colour space while MATLAB uses [0,360], [0,1], [0,1] for its HSV colour spaces ranges, respectively.

HSV colour space is known to be more suitable for image segmentation, because H and S components in general are not affected by light intensity and by the use of some tolerance value range, important traffic sign colours (Pixel of Interest) can be easily detected.

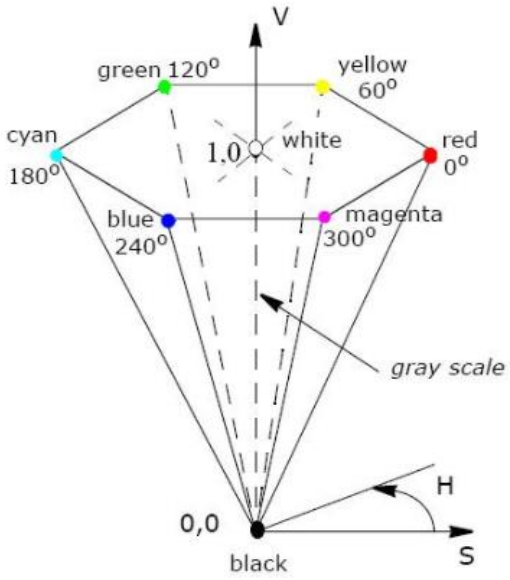

Figure 5.2 HSV Colour Gamut [42]

5.1.3 ROI proposal methodology

The process of discovering the ROI through colour segmentation is explained in the steps below

Step 1: Transform input RGB image to HSV colour space
Step 2: Segment at particular colour range
Step 3: Convert segmented image to binary
Step 4: Use Connected Component Analysis (CCA) to find ROI
Step 5: If ROI is available, crop image to ROI, return ROI as detected region and also return the co-ordinates of the detected region with respect to the original image

5.1.4 Video stream transformation from RGB to HSV and binarization

Since the video feed that will be fed to the Embedded Vision System is domiciled in the RGB colour space and segmentation is more stable and accurate in the HSV space, it is imperative to transform each frame of the streamed video into the HSV space before further analysis. The steps involved in the conversion of the video feed from RGB colour space to HSV are shown below. It is carried out based on (1) to (8) below after normalizing image to [0, 1] range (divide by 255). Post normalization, the operations mathematically stipulated by (4) to (9) are then carried out and used to derive the HSV based image.

$$Normalized\ Red\ (R') = R/255 \qquad (1)$$

$$Normalized\ Green\ (G') = G/255 \quad (2)$$

$$Normalized\ Blue\ (B') = B/255 \quad (3)$$

$$Min = min(R', G', B') \quad (4)$$

$$Max = max(R', G', B') \quad (5)$$

$$Range = Maximum - Minimum \quad (6)$$

$$Value = Maximum \quad (7)$$

$$Hue = \begin{cases} 0^o, if\ max = min \\ 60^o * \frac{G\prime - B\prime}{Range} + 0^0, if\ max = R' and\ G' \geq B \\ 60^o * \frac{G\prime - B\prime}{Range} + 360^0, if\ max = R' and\ G' < B \\ 60^o * \frac{G' - B\prime}{Range} + 120^0, if\ max = G' \\ 60^o * \frac{G' - B\prime}{Range} + 240^0, if\ max = B' \end{cases} \quad (8)$$

$$S = \begin{cases} 0, if\ max = 0 \\ \frac{Range}{max}, otherwise \end{cases} \quad (9)$$

5.1.3.2 Segmentation Relevant Colours from Image

To segment relevant colours from the image, the threshold values shown in Table 5.0 below are used. The HSV values generated are normalized to the range [0,1] before this threshold is applied. In other words, Hue values generated in OpenCV are divided by 180 while those generated in MATLAB by 360. Red colour can be found around two different hue location in the HSV space, and this is also well catered for in the thresholding.

Table 5.0 Table Showing Segmentation Threshold Values

	RED		YELLOW	BLUE
HUE	0.8<H<1	0<H<0.05	0.1<H<0.14	0.52<H<0.64
SATURATION	0.48<S<1	0.48<S<1	0.4<S<1	0.7<S<1
VALUE	0.4<V<1	0.4<V<1	0.8<V<1	0.1953<V<1

All pixels detected within the specified HSV range are labelled with a 1 while others labelled 0 hence creating a binary (black-and-white) image.

5.1.3.3 Performing Connected Component Analysis

Considering the fact that in the binary image, all relevant colours detected have been represented by a 1 and irrelevant colours 0, hence, blank-and-white, the Union Find algorithm employed in Maximally Stable Extremal Region (MSER) object detection and Connected Component Analysis is used to estimate possible ROIs. First of all, a sliding window of customizable window and step size is used to scan through regions of the mage. All the detected 1's are summed up to produce a Region Sum (RS). To determine if a region is a relevant area of the image, a 'beacon' value is used for which if the RS is greater than or equal to it, that region is highlighted to be a potential region. For this project, a beacon value equivalent to 1/20th of a 20 x 20 sized scanning window is used. In other words if 20 out of 400 pixels in a window is required to be a 1, before that region can be recognized as a potential ROI.

At the end of the analysis, the region with the highest region sum is selected and merged with nearby potential regions. Potential regions far away from the main region are discarded. Through this method, a most probable potential region of interest in discovered.

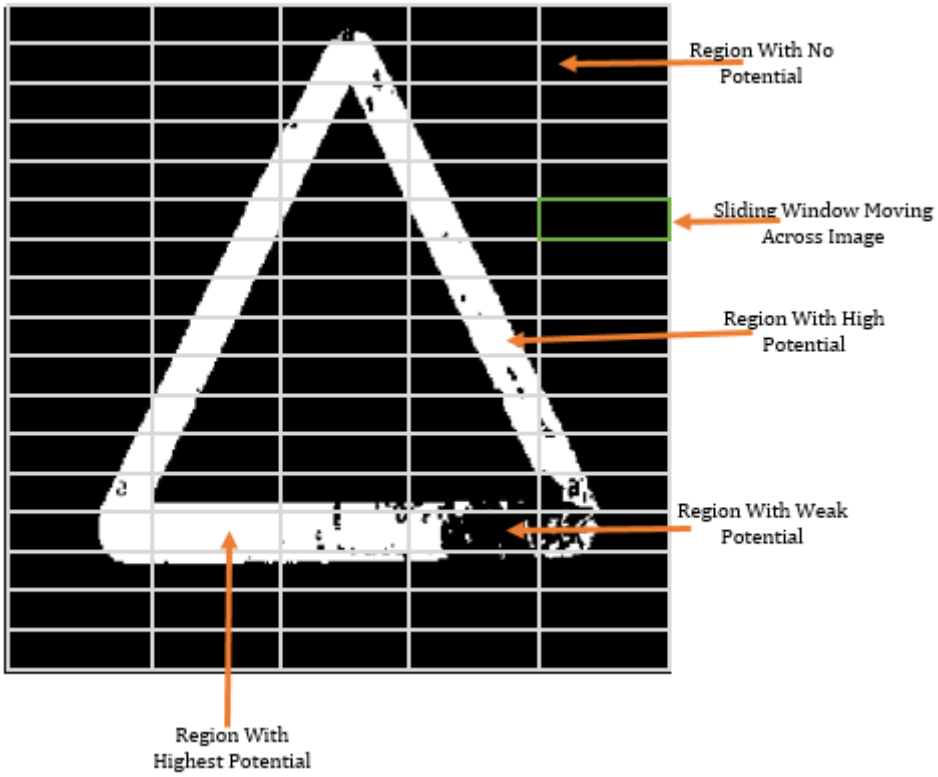

Figure 5.3 Figure showing Region Detection for a Potential 'Danger' Sign Using Colour Segmentation

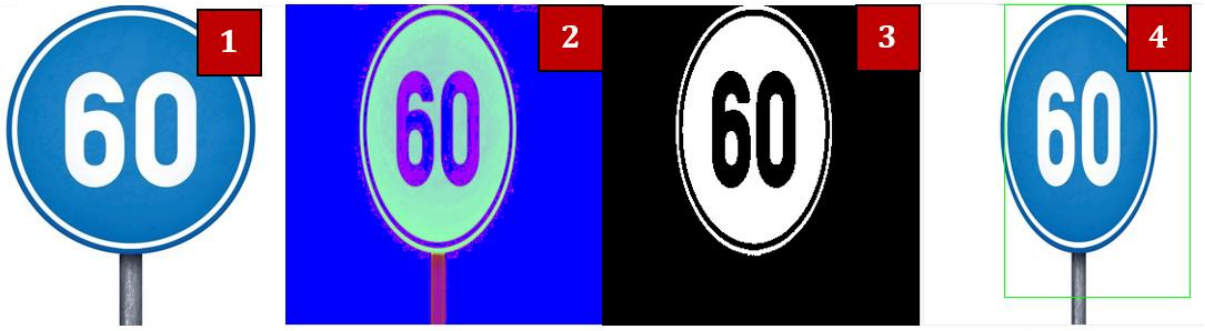

Figure 5.4. Region Detection Process Through Colour Segmentation from RGB-to-HSV Conversion, to Thresholding and Then Connected Region Location

Code B.3 of Appendix B.5 below shows how the complete CCA and ROI proposal is achieved using a combination of MATLAB and C++.

5.2 Second layer ROI validation (based on shape analysis)

As we know, basing our ROI judgement solely on colour inspection is not reliable due to the possible presence of objects with similar colours in traffic scenes. For this project 4 major traffic shapes are required to be detected. These are mainly circle, triangle, octagon and diamond shapes. Also square/rectangle detection is catered for.

The triangular signs are usually used to represent alert, circular for compulsory, rectangle and diamond for information. The region of interest that is obtained from colour segmentation explained above can now be validated depending on their shapes by employing various shape analysis algorithms such as Circular and General Hough transform [43], Sobel edge detection methodology, Harris corner detection algorithm and Paulo sub-region analysis algorithm. Different combinations of these algorithms are used to tackle the detection of each of these shapes in the proposed ROI.

5.2.1 Circle detection/validation

In order to detect the presence of a circle, edge detection is first of all carried out using Sobel Edge Filter. The Sobel operator performs a 2-D spatial gradient measurement on an image and so by so doing, emphasizes regions of high spatial frequency that correspond to edges. Typically it is used to find the approximate absolute gradient magnitude at each point in an input grayscale image [44] and to simply put, edge detection algorithms help us determine and separate objects from background in images [45].

The Sobel Edge Detection (SED) method makes use of primarily two filters which together when convolved with a part of an image will produce the image gradient in that particular area (an image's gradient represents a change in its intensity or colour). Interestingly, a smooth correlation exists between edges and gradients, as we now know that edges occur in images when the gradient is greatest. The operators of the SED method make used of this paradigm to detect edges in images.

5.2.1.1 Sobel Edge Detection Methodology

To achieve the edge detection, two Sobel filters G(x) and G(y) are convolved with the image, resulting in the derivation of the gradient of the image both in the horizontal and vertical directions. The magnitude of the gradient can then be computed by summing the two (x and y) gradients together.

$$G(x) = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix} \qquad (10)$$

$$G(y) = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \qquad (11)$$

The magnitude of the gradient is given by (12) below

$$|G| = \sqrt{G(x)^2 + G(y)^2} \qquad (12)$$

$$\theta = \arctan\left(\frac{G(y)}{G(x)}\right) \qquad (13)$$

The steps listed below explain the 6-stage process of performing edge detection on the received video stream

Step 1: Check to know if image is coloured (Input should be an RGB image)
Step 2: If coloured, convert to gray scale
Step 3: Convolve Sobel filter with image
Step 4: Get gradient magnitude for each completed convolution
Step 5: Create edge image using gradient magnitude
Step 6: Perform max and median filtering

Median filter is applied by subtracting the median value from all pixels in the edge image while the max filtering is achieved by blacking out all pixels who's magnitude value are

less than 30% of the maximum/brightest pixel in the edge image. Code 7 below gives more details on the edge detection process using the Sobel filter.

5.2.1.2 Circular Hough Transform (HT)

The Hough transform developed by Paul Hough in (1962), Richard Duda and Peter Hart (1972) is a feature extraction technique used in image analysis, computer vision, and digital image processing [46]. It can be used to determine the presence of shapes and lines in an image through a voting procedure. In particular, the Circular Hough Transform which was later discovered in 1992 can be used to determine the presence of a circle once the perimeter's pixel locations are known. It uses the equation of a circle as shown in (14) below. This explains why the edge detection framework (SED) is first of all used so as to keep only the circle's edges/perimeter showing rather than its entire body (resulting in lesser computation).

$$r^2 = (x-h)^2 + (y-k)^2 \qquad (14)$$

(h, k) – Circle Centre

(x, y) – Pixel point on the circumference of the circle

r - Radius of the circle.

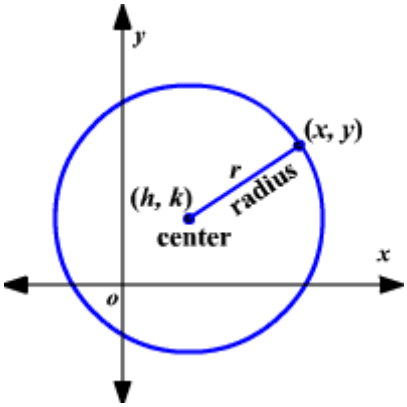

Figure 5.5 Figure Illustrating the Berhman Circle Equation

Parametrically, the circle is represented by (15) and (16) below

$$x = h + r\cos\theta \qquad (15)$$

$$y = k + r\sin\theta \qquad (16)$$

This specifically means that, given the various coordinates (x, y) which make up the circle's perimeter, multiple centres (h, k) can be found by varying the radius r and angle θ and thereafter checking for the most occurring value of h and k. The fact that the Circular Hough Transform relies on 3 parameters makes it even more computationally and memory expensive and its performance is highly dependent on the efficiency of the edge detection algorithm, however, it is still one of the best available algorithms.

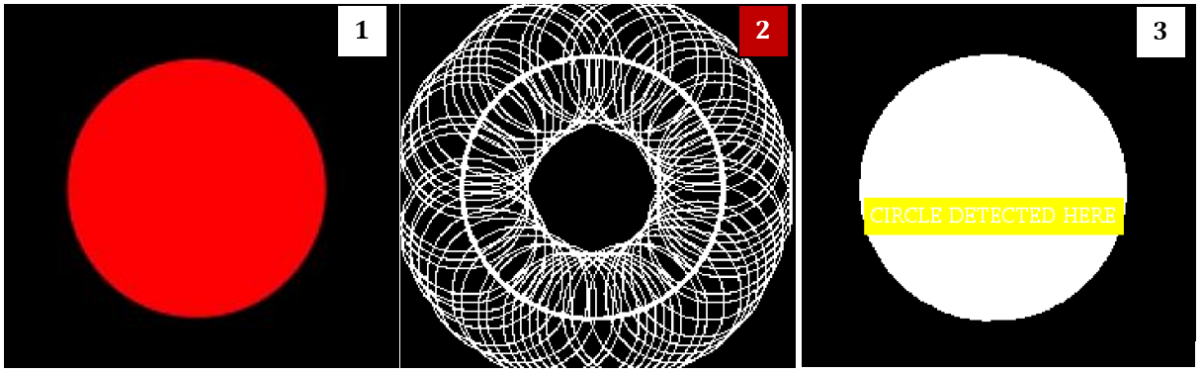

Figure 5.6 Figure Showing Code Outputs (2, 3) During Circle Detection Process for Input Image (1)

The steps below explain in a broader manner, the Circular Hough Transform Algorithm

Create Accumulator [h,k,r]
For each pixel (x,y)
for minRadius to maxRadius
for $\theta = 0\ to\ 360$
$$a = ceil(x - rcos\ \theta)$$
$$b = ceil(y - rcos\ \theta)$$
$$Accumulator[a,b,r] = Accumulator[a,b,r] + 1;$$
end for θ
end for minRadius
End for each pixel

5.2.2 Triangle/octagon/square and diamond detection/validation

Other classes of traffic sign shapes to consider are the triangular, octagonal, square and diamond shaped signs. To detect the presence of these shapes the Line Detection Hough Transform Method is used in this project. In total, two detection methods are compounded and used to validate the detection of a triangle, square, octagon and diamond shaped sign. First off is the Line Detection Hough Transform method which scans the pixels to determine the presence of straight intersecting lines matching the pattern of the shape being searched for and then an additional sub-region validation method found in [47] is used to complement this.

5.2.2.1 Hough Transform (HT) line detection method

In general, any straight line given by (16) below can be represented as a point (c, m) in the parameter space. However, vertical lines which have infinite slopes cannot be accurately represented this way. For this reason, Duda and Hart proposed the use of the Hesse normal form whereby equation (17) below is substituted in place of (16).

$$y = mx + c \qquad (16)$$

$$\rho = x\cos\theta + ysin\theta \qquad (17)$$

In (17), ρ represents the distance from the origin to the pixel in question while θ is the angle between the x axis and the line connecting the origin with that closest point as shown in Fig. 16 below

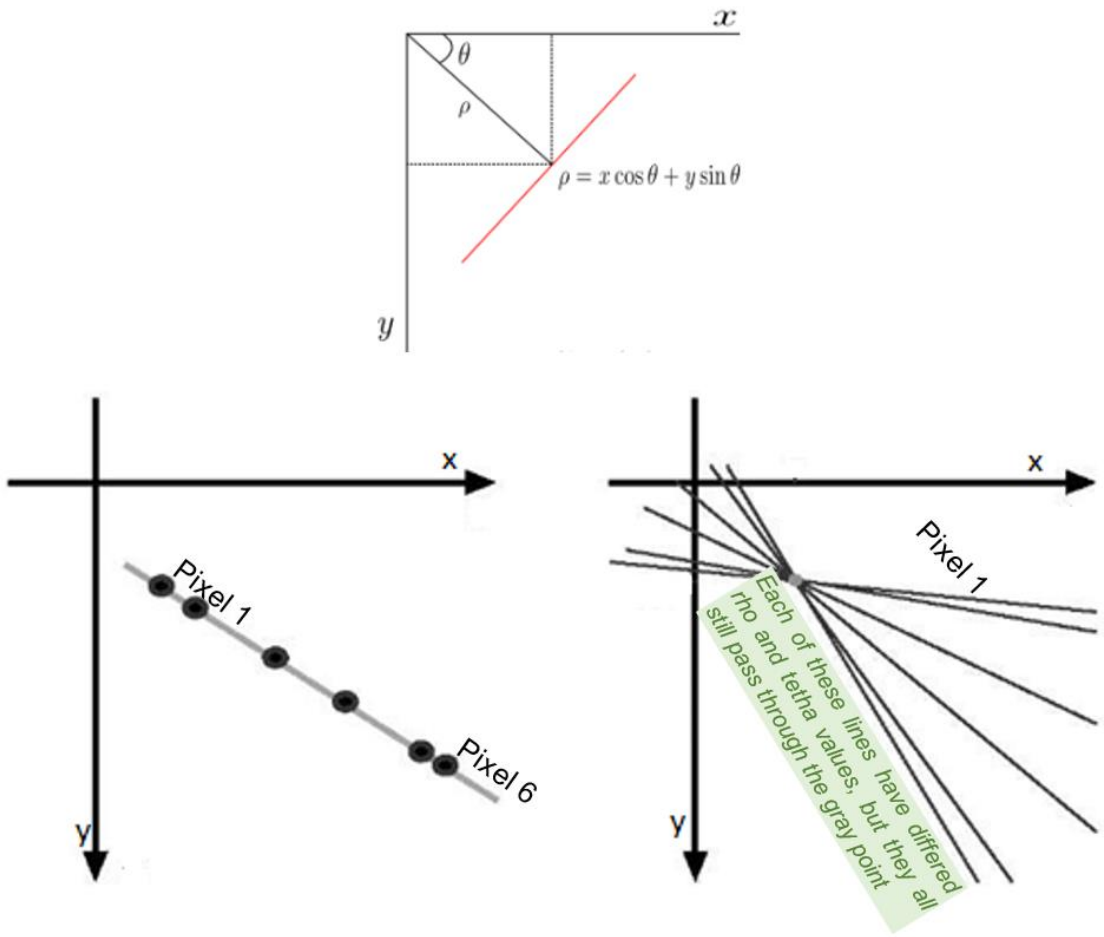

Figure 5.7 Figure Explaining the Concept Behind the Hough Transform

Recall that a straight line will have many pixels on it as shown in Fig. 16 above. Therefore, visiting each pixel in the image left behind by the edge detector and drawing many arbitrary line paths through each (mostly by varying θ between -90 and 89 degrees), multiple (ρ, θ) associated with each pixel (x, y) in the image can be derived and stored. If two pixels are on the same line, it means they share at least one (ρ, θ) value in common asides other (ρ, θ) values associated with them. Therefore, for a straight line with many pixels on its path (e.g. 200 pixels), there will be at least 200 pixels associated with a particular (ρ, θ) value. By getting reasonably many enough (ρ, θ) associated with each pixel in the image, we can search the parameter space containing all these (ρ, θ) and confidently assume that all the (ρ, θ) which are associated with numerous enough pixels are lines. The steps listed below explain how this Hough line detection methodology is used in the detection of triangular, octagonal, square/rectangular and diamond shapes

Step 1: Get Image and convert to grayscale
Step 2: Perform edge detection on image
Step 3: Create Hough parameter space, a 2D array (ρ, θ) with maximum ρ length spanning the diagonal length of the image and maximum θ spanning 180.
Step 4: Initialize Hough parameter space to zero
Step 5: Loop through entire image and for every active pixel (1, not 0) on co-ordinate x, y, calculate as many (ρ, θ) values associated with it by varying θ in (17) above between -90 and 89.
Step 6: For every pixel which has a line lying on (ρ, θ), increment the value stored in

(ρ, θ) by 1

Step 7: After looping through all the relevant pixels of the image and incrementing (ρ, θ) appropriately in the Hough parameter space, thereafter search space for the peak values.

Step 8: The (ρ, θ) accommodating this peak values can be used to draw the detected lines.

Step 9: Find the angles between detected lines.

a. If three or more lines have angles of 60 degrees with each other, then a triangle is most likely present
b. If eight or more lines that have angles 135 degrees exist, then an octagon is most likely present
c. If four or more lines exist that have angles 90 degrees, then a square, rectangle or diamond is most likely present

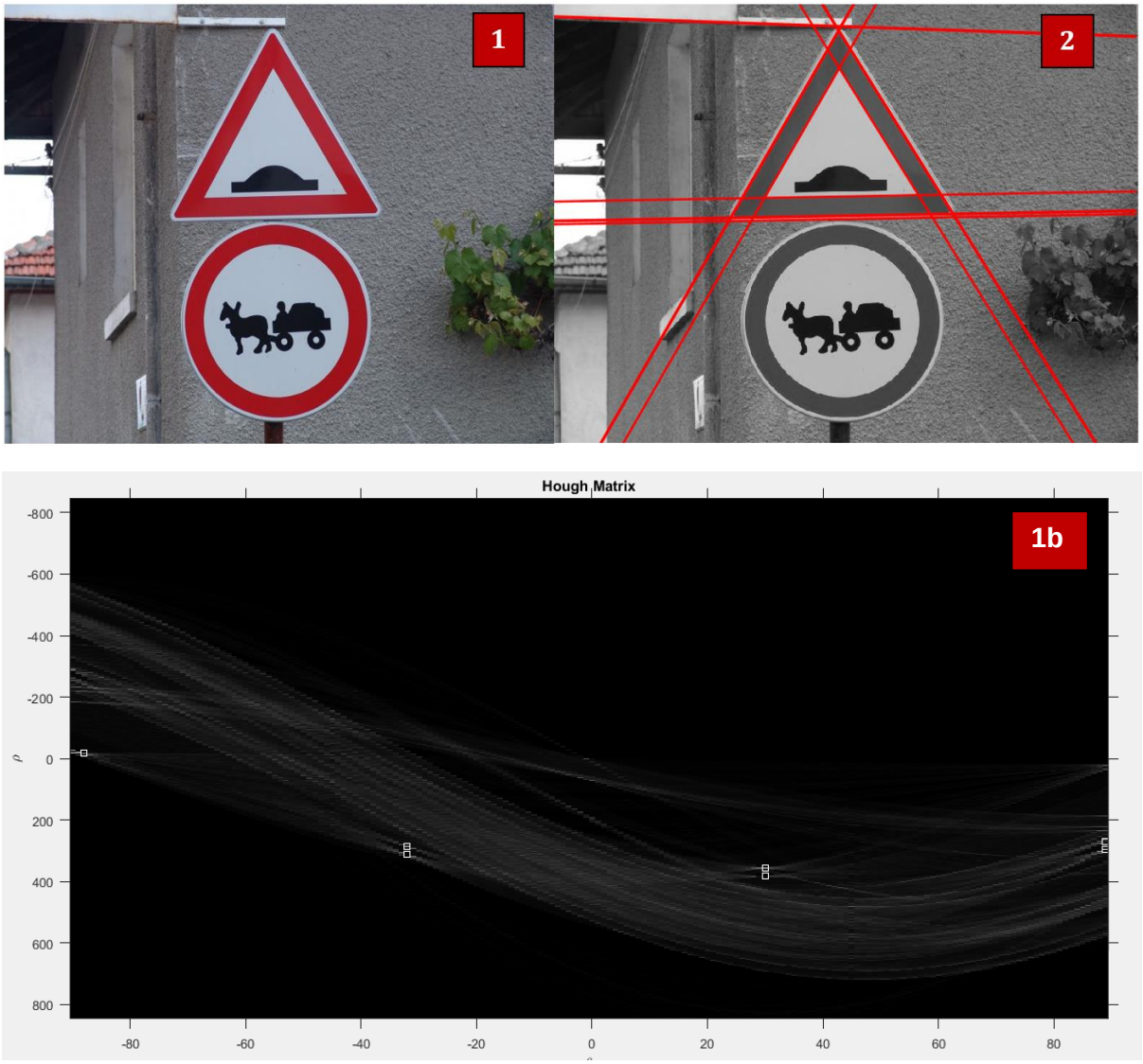

Figure 5.8. Figure Showing Triangular, Square, Octagonal, Rectangular and Diamond Shape Detection Performed On Traffic Sign Scene Using Hough Transform and A Plot of the Hough Parameter Space Showing Detected Peaks

See Appendix B for more details on the codes written for the practical implementation of the Hough transform for shape detection in MATLAB, C++ and OpenCL

5.2.2.1 Sub-region analysis using Harris corner detection

A top level method is used to validate the presence of triangles and squares in the proposed region of interest. This is set in action by using the Harris corner detector methodology to isolate only peaks in the ROI. Further to this a search is conducted at the top left and right, top centre as well the bottom left and right and bottom centre sub-regions, so as to confirm the presence of up pointing, down point triangles or squares.

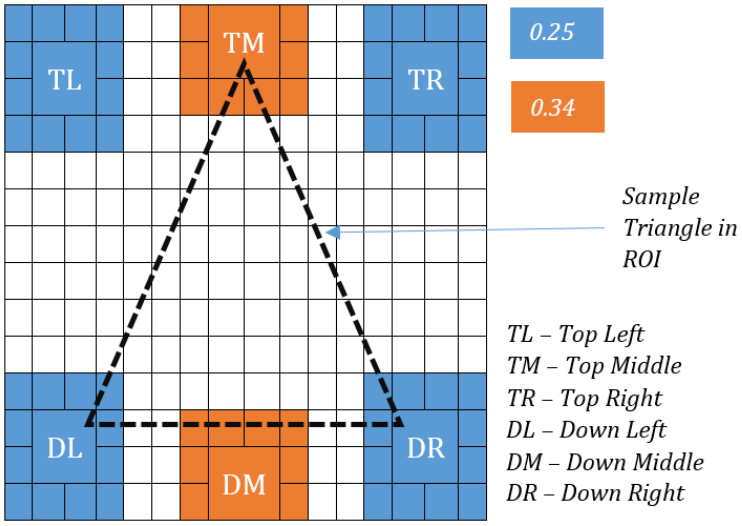

Figure 5.9. Figure Showing Second Shape Validation/Analysis Mechanism

After a scan of each of the sensitive sub-regions within the proposed ROI i.e TL, TM, TR, DL, DM and DR is carried out, for each region wherein a corner detected, appropriate region weigh values (0.25 or 0.34) are assigned to them. Finally, (18), (19) and (20) [46] are used to determine if the object present in the detected ROI is an upward/downward facing triangle or a square/rectangle.

$$Square = TL + TR + DL + BR \quad (18)$$

$$Triangle\ (Upward) = 1.32 \times (DL + DR) + TM - 1.1 \times (TL + TR) \quad (19)$$

$$Triangle\ (Downward) = 1.32 \times (TL + TR) + DM - 1.1 \times (DL + DR) \quad (20)$$

The shape from equation (18), (19) and (20) above with the highest percentage is selected as present. See Appendix B for more details on the codes for critical shape analysis.

As we know, triangles typically have 3 corner. For upward facing triangles, the apex is at the top while the down left and down-right corners bare the remaining two. This distinctive feature of triangles comes in handy in its detection (similar for squares also). The Harris Corner Detection methodology introduced by Stephen and Harris as an improvement to Moravec's corner detection method is used in this project to detect points of change in image intensity. The Moravec's which was being earlier used in Computer Vision determined the average change of image intensity from shifting a small window across different parts of the image. Harris improved this by using gradient formulation from Gaussian distributed weight values to detect response at any shift. In summary, the Harris Corner Detection glides over the image, looking for x, y values whereby movement of the window produces a noticeable gradient change. For the sub-region analysis, the Harris Corner detection is first of all implemented. This sifts out the 3 triangular corners or 4 square/rectangular corners. The corners detected are fed to the sub-region analyser highlighted above to validate the presence of these shapes.

A corner can be defined as the intersection of two edges. It can also be defined as a point for which there are two dominant and different edge directions in a local neighbourhood of the point. One determination of the quality of a corner detector is its ability to detect the same corner in multiple similar images, under conditions of different lighting, translation, rotation and other transforms.

The Harris corner detector makes use of (21) below and can be further broken down into (22) and (23)

$$E(u,v) = \sum_{x,y} w(x,y)[I\big((x+u),(y+v)\big) - I(x,y)]^2 \quad (21)$$

$$M = \sum w(x,y)\begin{bmatrix} {I_x}^2 & I_x I_y \\ I_x I_y & {I_y}^2 \end{bmatrix} \quad (22)$$

$$E(u,v) = [u\ v]M\begin{bmatrix} u \\ v \end{bmatrix} \quad (23)$$

$$E - Difference\ between\ original\ and\ moved\ image$$
$$u - Windows\ displacement\ in\ the\ x\ direction$$
$$v - Windows\ displacement\ in\ the\ y\ direction$$
$$w(x,y) - Window\ at\ position\ x,y$$
$$I - Intensity\ of\ the\ image\ at\ position\ x,y$$
$$I(x+u,y+v) - Intensity\ of\ the\ moved\ window$$
$$I\ (x,y) - Intensity\ of\ the\ original$$

The steps below explain succinctly how the Harris Corner Detection is carried out

Step 1: Load Image
Step 2: Compute First Order Derivative of Image
Step 3: Compute Second Order Derivate of Image
Step 4: Create Isotropic Window
Step 5: Compute Trace and Determinant
Step 6: Compute Harris Response
Step 7: Use Non-Maximal Suppression to Sift in the Best Edges

Loaded image should be converted to gray scale and then the first order derivatives of the image in both horizontal and vertical directions is computed. To achieve this, a filters (dy = [-1 0 1; -1 0 1; -1 0 1]) for horizontal derivative and its transpose (dx = ‘dy) for the vertical derivative are convolved with the image. To get the second order derivative, a simple element wise multiplication of the first order derivatives is carried out.

To check the cornerness of each pixel of the image, a small isotropic area around that pixel is checked using a smooth circular window. To do this, we filter all three second-order derivatives by convolving them with a Gaussian kernel. The filtered second order derivative of the image $({I_x}^2, {I_y}^2, I_x I_y)$ are used to compute the unique value called the Harris Response (R). The Harris response for each pixel in the image is used to determine whether a pixel is a corner or not through non-maximal suppression and thresholding. In doing this we assume that each pixel can only be a corner if it is a maximum compared to its 8 adjacent pixels.

$$R = \det(M) - (k * (trace(M)^2) \quad (24)$$

$$trace(M) = {I_x}^2, + {I_y}^2 \quad (25)$$

The value k was empirically computed and given by Harris to be around 0.004 and 0.06

By thresholding the response of the non-maximally suppressed image with the minimum desired Harris response, we are therefore able to sift out only parts of the image whereby only pixels with a value greater than this threshold values are. For this project, a k value of 0.04 is used, Harris Threshold of 1000000 and suppression window size of 20.

Figure 5.10. Figure Showing Detected Corners of a Triangular Sign

5.3 Vision System's detection mechanism performance testing

Considering the fact that a videos are simply multiple frames of an image streamed per second, to test and evaluate the performance of the designed system, individual frames of a 2 minutes 30 seconds long video from [48] of a car driving around a city are fed to the system, and the performance recorded are as highlighted in the table 5.1 below.

Table 5.1 Table Showing Result of System Test and Evaluation

COMPUTER DETAILS RAM: 16GB OS: 64bit Windows OS Processor: Intel ® Core TM i5 Processor Speed: 2.70 GHz									
	Video Details			**Performance Evaluation**					
DETAILS	***Image Size***	***Video Duration***	***Number of Available Traffic Signs***	***Number of Traffic Signs Detected***	***Accuracy***	***False Detection***	***Potential Region Predicted (Right)***	***Potential Region Predicted (Wrong)***	***Speed (s/frame)***
Video Test 1	500 X 500	2 Minutes, 30 Seconds	25	25	100%	12	9	2	1.525s
Video Test 2	300 X 300	2 Minutes, 30 Seconds	25	24	96%	6	8	1	0.625s

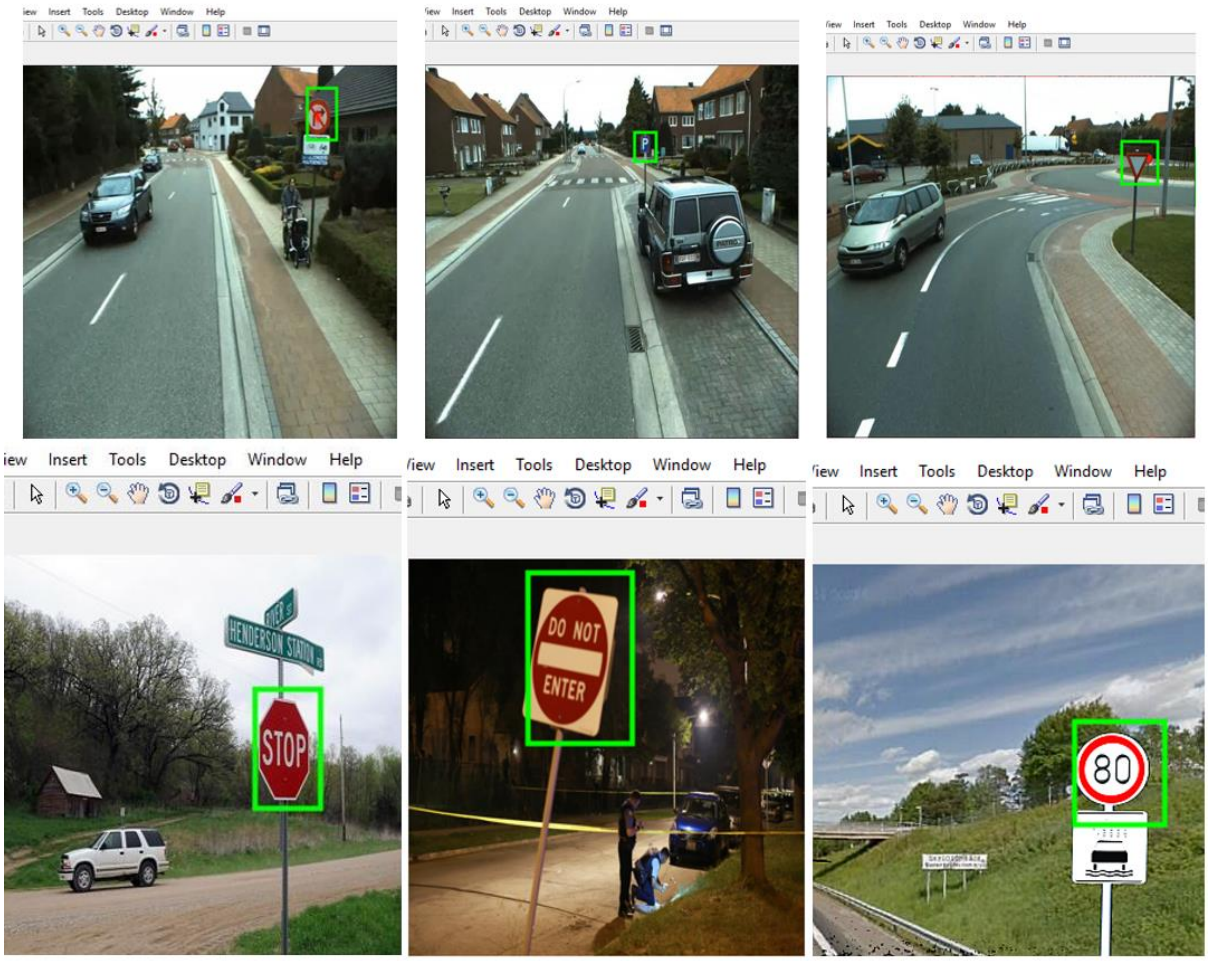

Figure 5.11. Figure Showing Project Traffic Sign Detection

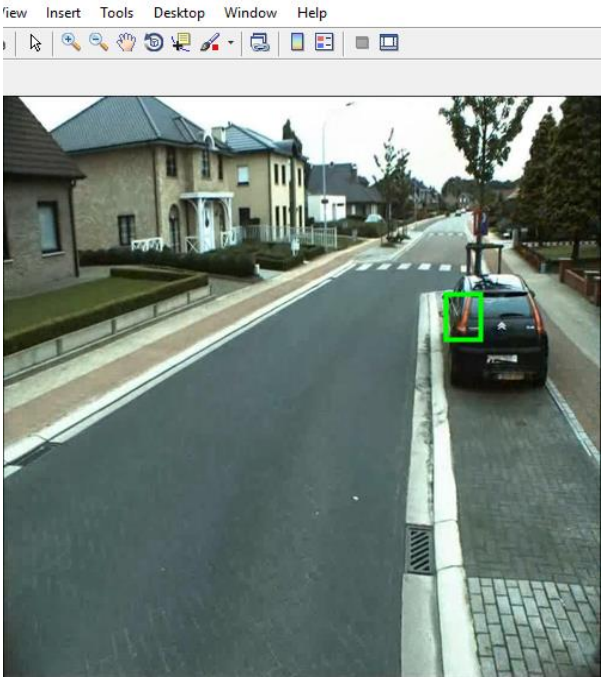

Figure 5.12. Figure Showing an Example False Detected Sign During the System Test

See Appendix B for more details on the test codes written to test the performance of the system's traffic sign detection module.

5.4 The optimization challenge

At this point, the computer vision system's traffic sign detection and classification modules have been designed, implemented, tested and evaluated on CPU, using MATLAB, with no form of parallel computing methodology employed, nor GPU arrays used or any other form of performance optimization mechanism utilized. All detection and classification functions are executed in series.

Overall, a frame analysis speed of about 1.5 seconds per frame was attained for detection and sign classification. While this exists, the last phase of the project attempts to utilize various software and hardware optimization approaches to speed up the performance of the system and tries to evaluate the level of improvements achieved due to each of the optimization improvement techniques utilized.

6.0 Vision System's Performance Evaluation, Optimization, and Analysis

At this stage, the classification and detection system have been well integrated and tested, with less consideration given to code optimization, and performance improvement. This chapter helps lay a good foundation on how the achieved performance shown in Table 5.1 of section 5.3 above can be improved via various methods such as code optimization as well as parallel and heterogeneous computing. With heterogeneous computing, the vision system simultaneously makes use of different kinds of compute devices (CPU, GPU and FPGA) for different operations needed to be carried out.

The key optimization schemes used in the improvement of the performance of this system include:

1. MATLAB Vectorization
2. Use of C/C++/FORTRAN for some subroutines
3. MATLAB Parallel Computing
4. Heterogeneous Computing

A good part of the rest of this chapter explains why and how each of these optimization is achieved and also shows with charts, the percentage improvement recorded.

6.1 Optimization through vectorization (MATLAB)

Most modern CPUs are designed to be able to carry out single instructions with multiple data, thereby making them perform same operation on multiple data points simultaneously. This form of operation is very important when dealing with digital media content on CPU. With this, same instructions that need to act on different pieces of data can be computed simultaneously, thereby leading to an increase in speed.

Rather than using two nested for loops to scan through and select pixels of a 2D image (img) in MATLAB, a vectorised approach can be used (e.g img(:)) to do this. Going through the entire project codes and making effort to vectorise as many as possible image interaction instructions lead to a great deal of improvement in the speed of the detection system. Figure 6.0, Table 6.0 and 6.1below shows the various time improvement gained in the execution of some of the vision system's key functions.

6.2 Optimization through the use of C/C++ for some subroutines

Although MATLAB is known to be very efficient for tasks which can be vectorized, it also has its own deficiency, especially with for loops, as MATLAB is originally built to operate on matrices. Considering the fact that there exist some part of the detection operation used in this project which are of the complexity order 0(n^3), some of the detection subroutines are therefore written in C/C++ and then called from MATLAB using the MEX gateway function. (See example in Appendix B.5). Two important functions were highly optimized using this method. The Connected Component Analysis function improved in speed by ***508 times*** and the 'Shape Analyser Function' improving in speed by over **25 times** and over **17 times** improvement in speed for the edge detection function. The MATLAB Vectorized version of the other functions had better performance (see figure 6.0). The C++ code for the CCA function is written using visual studio and linked using the MATLAB MEX gateway. For the other functions, the MATLAB code generator is used to generate their C/C++/FORTRAN equivalent.

All test are performed using the computer with the details below

Table 6.0 Table Comparing Performance of the Various Subroutines Based on Different Optimization Scheme Employed. Detection time is in s/frame

COMPUTER DETAILS **RAM:** 16GB **OS:** 64bit Windows OS **Processor:** Intel ® Core TM i5 **Processor Speed:** 2.70 GHz **Image Size:** 500x500							
S/N	**Detection Function**	**Function Description**	**Un-Optimized MATLAB (s/frame)**	**Optimized MATLAB (s/frame)**	**Improvement (MATLAB un-optimized vs optimized)**	**C++ (s/frame)**	**Improvement (MATLAB un-optimized vs C/C++)**
1	*analyseCriticalAreas.m*	Uses the Harris corner detector methodology to isolate only peaks in the ROI and then checks for the presence of rectangles and triangles.	0.080	0.0935	Optimized version, 1.16 times slower	0.107	C/C++ version, 1.33 times slower
2	*CCA.m*	Connected Component Analysis function makes use of a sliding window to detect the presence of a potential ROI	0.349	0.0724	Optimized version, 4.8 times faster	0.000687	C/C++ version, **508 times faster**
3	*detectCircle.m*	Used to Detect Circles in the ROI	0.301	0.114	Optimized version, 2.64 times faster	0.167	C/C++ version, 1.8 times faster
4	*myDetectCircle.m*	Customized second level circle detection function					
5	*detectEdge.m*	Used to detect the presence of edges in the ROI	0.269	0.0203	Optimized version, 13.25 times faster	0.0157	C/C++ version, **17.1 times faster**
6	*detectCorner.m*	Used to detect the presence of corners in the ROI	0.158	0.128	Optimized version, 1.23 times faster	0.135	C/C++ version, 1.17 times faster
7	*classifyImage.m*	Function used to interface with the MATCONVNET image classification library	-	0.0164	-	-	-
8	*shapeAnalyser.m*	Used to Detect the presence of rectangles, triangles, octagons and diamond shaped shapes	0.393	0.044	Optimized version, 8.93 times faster	0.0152	C/C++ version, **25.88 times faster**

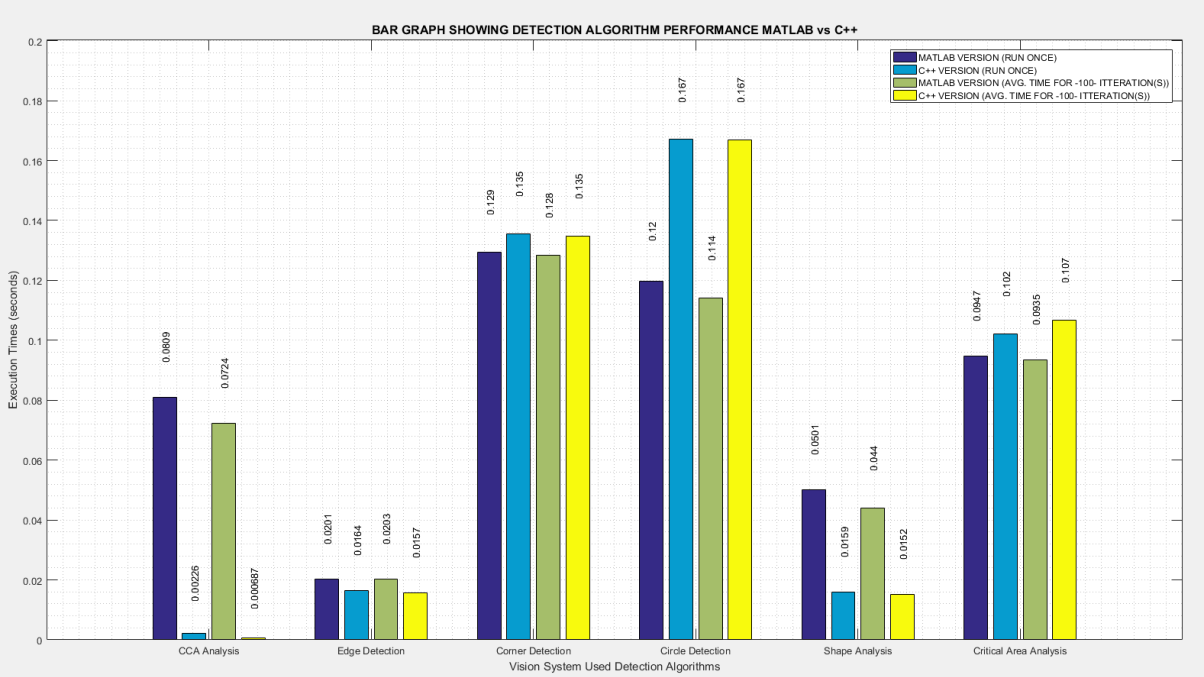

Figure 6.0 Bar Chart Showing Average Speed of the Various Detection Algorithms Used and Comparing Each's Performance in MATLAB against C/C+

6.3 Optimization through parallel execution

Parallel computing is a type of computation in which many calculations or the execution of processes are carried out simultaneously [49]. This is most feasible when different threads/tasks within a program can be executed independently without any form of dependencies between each other. In cases like this, computers with many cores can pass each of these tasks to their different cores for execution simultaneously. For this project's computer vision system, there exist some independent task such as circle detection and shape analysis which have no shared dependencies. MATLAB parallel computing toolbox is used to fire the execution of these two functions (in parallel) while other serial interdependent parts of the programme keep running. In other words, the group of serially executed functions are executed concurrently with the independent functions. By the end of the execution of the group of serially dependent functions, the other independent functions being run in parallel will also have been completed, thereby increasing holistic detection time for a single image frame.

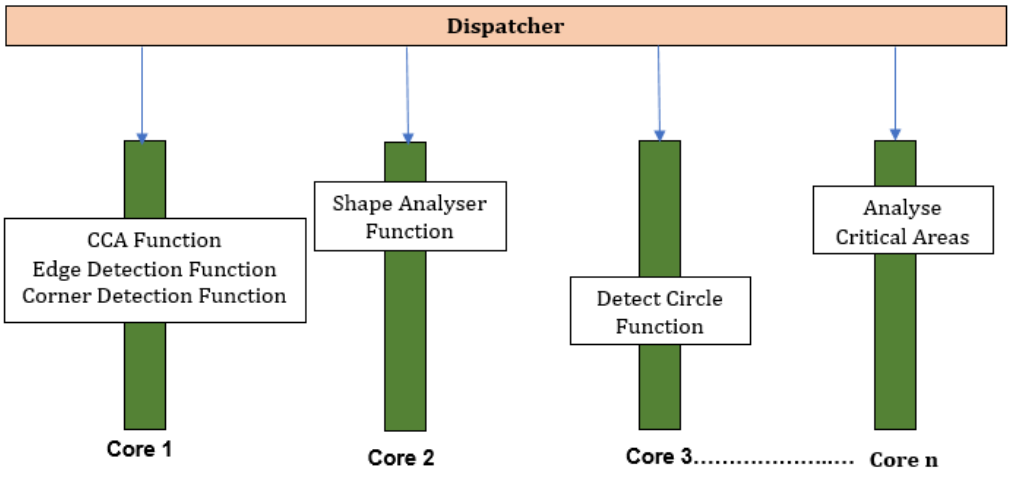

Figure 6.1 Figure Illustrating the Parallel Execution Approach for Detection

Table 6.1 Table Comparing Detection Time Per Frame Performance of the Various Subroutines Based on Different Dispatch Method

	MATLAB (Vectorized) + Serially Dispatched	MATLAB (Vectorized) + Parallel Dispatched	C/C++ (Serially Dispatched)	C/C++ (Parallel Dispatched)
Detection (frames per sec)	0.11922		0.02355	

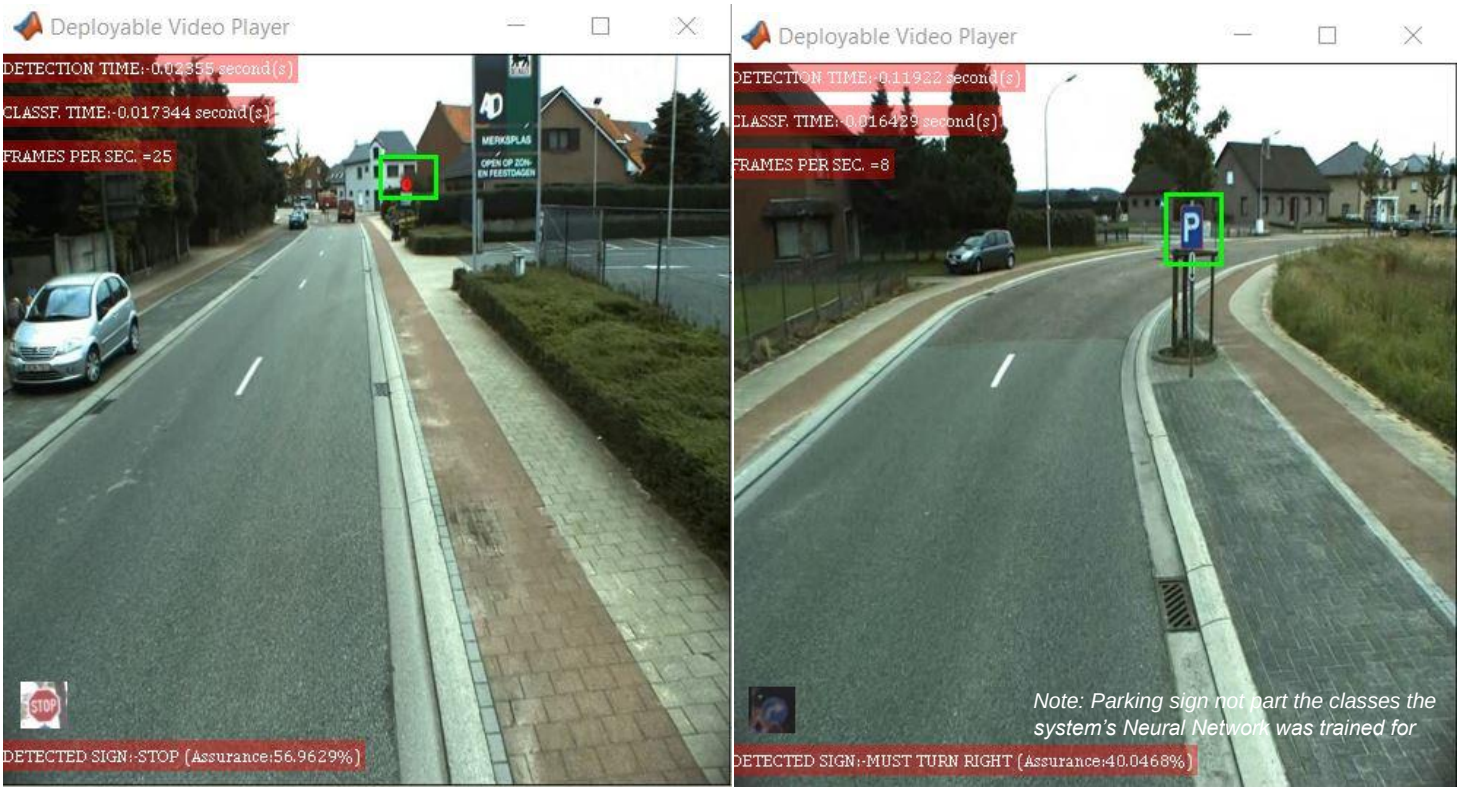

Figure 6.2 Figure Showing Real-Time Vision System Output When Fed With Live Traffic Scene Video Input

6.4 Optimization through heterogeneous computing (High Level Synthesis)

Heterogeneous computing has become a stable way through which computationally intensive Computer Vision task are now executed. As we know, CPUs are best for arithmetic tasks while GPUs do best with graphics related computations. Also, Field Programmable Gates Array (FPGAs) are even more unique, as their reconfigurable circuitry can be configured to model (in hardware), software operations. With Heterogeneous computing the best characteristics of these kinds of compute devices are leverage and used together in one system to provide a very fast and effective performance. In essence, to optimize the performance (speed) of the computer vision system in this project, the host PCs GPU (if present) is sanctioned with the task of performing the computationally intensive circle identification operation of the traffic sign detection.

To test the efficacy of these method, a custom built circle detection function is written in MATLAB and also replicated as an Open Computing Language (OpenCL) based

application, so that its performance can be tested on an FPGA or GPU. OpenCL allows software and hardware programmer write codes targeted at specific hardware devices such as GPUs and FPGAs. Often, OpenCL applications run on a host CPU, but are able to call various other connected computing devices attached to them to assign task (see code 6.0 in section 6.4.2). The performance of the OpenCL based application is measured on FPGA as well as GPU and benchmarked against normal CPU performance with MATLAB.

A custom written circle detection function is used because of the complexity involved in trying to rewrite MATLAB's 'imfindcircles.m' function which is known to call several other sub function also. Therefore for accurate comparison, a new 'myDetectCircles.m' function is written (in MATLAB) based on the Circular Hough Transform (CHT) theory and also in C/C++ before being resolved into an OpenCL application. It is this customized circle detection function whose performance is evaluated (MATLAB vs C++ vs OpenCL). Overall, performance improvement gleaned here can be projected as similar to what will be obtained if the MATLAB native circle detection function is also re-written as an OpenCL application. Being able to easily dedicate this task to the connected GPU/FPGA will greatly improve this projects computer traffic sign detection and recognition system's performance (speed).

For better understanding of the FPGA and GPU technology see Appendix A.1 and Appendix A.2. In the past, FPGA technology could be used only by engineers with a deep understanding of digital hardware design, however, the rise of high-level design tools has now changed the rules of FPGA programming, with new technologies that convert graphical block diagrams or even C code into digital hardware circuitry.

Engineers have now developed a higher-level tools which takes away the need to use Hardware Descriptive Language (HDL) which function by modelling the actual logic circuit. One of the frameworks developed in this domain is the Open Computing Language (OpenCL), initially developed by Apple Inc. A framework for writing programs that execute across heterogeneous platforms such as FPGAs, CPUs, GPUs and DSP chips. OpenCL provides a standard interface for parallel computing using task- and data-based parallelism. It is based on the C99 and C++11 programming language and most importantly, specifies an Application Programming Interfaces (APIs) to control the hardware platform and execute instructions on them. Third-party APIs exist for other programming languages and platforms such as Python [50] Java and .NET [51]. It is an open standard maintained by the non-profit technology consortium Khronos Group, Companies which have complied with this open standard include Altera, Nvidia, AMD, Apple, IBM, Samsung, Vivante, Imagination, Intel, ARM, Creative, Qualcomm and Xilinx [52][53].

OpenCL functions are normally referred to as 'kernels' [54] as the framework is designed to see computing systems as systems which consists of a number of compute devices e.g. CPUs, GPUS attached to a CPU [55] etc. Each of these compute devices are further broken down into multiple Processing Elements (PEs). OpenCL therefore uses a language closely related to C to write kernels which can be run on all or as many of the PEs in parallel. The subdivision of these individual compute devices into compute units and PEs is determined by the compute device designer. The OpenCL API allows programs running on the host to launch kernels on the compute devices and manage device memory.

For this project, the Intel OpenCL Software Development Kit (SDK) which provides all the necessary tools needed to build and run OpenCL-based applications on Altera FPGAs is used. Primarily, the host program of the OpenCL application is executed on the CPU, and the kernels are compiled into hardware circuits, placed into the FPGA, and launched on-demand by the host program. The SDK is made up of essentially three components which are:

- The Intel Offline Compiler (AOC)
- The Host Runtime
- The AOCL Utility

The AOC translates the OpenCL kernel code into hardware circuits which the FPGA can be reprogrammed after while the host runtime is a collection of libraries and drivers that allow for communication between the host programme and the kernels in the FPGA. The AOCL utility provides a set of commands to perform certain important operations such as performing diagnostic check on the OpenCL drivers and downloading the kernel into the FPGA.

The Intel OpenCL SDK in conjunction with a standard C++ compiler is used to compile the OpenCL application

Appendix A.3 to *A.9* as listed below gives more details on OpenCL for High Level Synthesis. Its operation, memory architecture, language structure, limitations etc.

- Appendix A.3: OpenCL architecture
- Appendix A.4: OpenCL execution model
- Appendix A.5: OpenCL Actual execution Pipeline
- Appendix A.6: OpenCL memory architecture
- Appendix A.7: OpenCL programming language structure
- Appendix A.8: Limitations in OpenCL
- Appendix A.9: Types of parallel programming

6.4.1 Accelerating circle detection task through OpenCL on FPGA

To test and understand how well off a performance improvement can be gotten through heterogeneous computing, the computationally expensive circle detection operation carried out by the vision system can be assigned to an FPGA. The steps listed below shows the needed procedure to be taken in the development of the OpenCL application for circle detection and Figure 6.3 below gives a diagrammatic explanation of the implementation process

1. Writes OpenCL application code which is made up of C/C++ host program and OpenCL kernels
2. Compile Host Programme using a C compiler e.g. GCC. This creates the executable binaries
3. Compile the kernels using the AOC. This creates the .aocx file
4. Use the AOCL utility to load the .aocx into the FPGA
5. Run the host programme which will from time to time launch the OpenCL kernels on the FPGA (Accelerator) when it needs to do so.

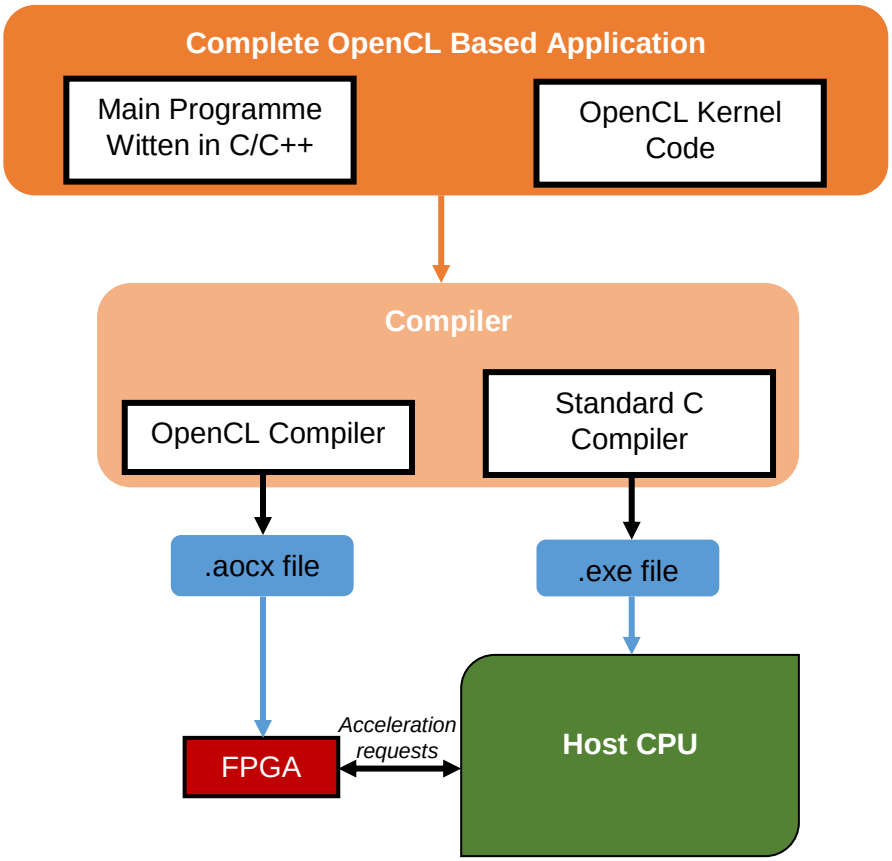

Figure 6.3 Diagrammatic Explanation of the OpenCL Application Implementation Process

The steps highlighted below shows on a granular level how the OpenCL kernel is executed on the FPGA at runtime, from the host platform

1. Host platform initializes OpenCL by querying available platforms
2. Desired platform is selected
3. Select desired device (CPU, GPU, ACCELERATOR etc.)
4. Create context to manage the selected device and kernel execution
5. Create command queue attached to the device context
6. Create memory objects attached to the context
7. In preparation for the kernel about to be launched, write all needed data from host memory onto the memory objects (created in the step above) which actually will reside in the selected device's memory
8. Load kernel file (.cl) containing one or more kernels to be launched
9. Create kernel program object (a collection of ready to run kernels) using the loaded kernel file
10. Build created program object into a binaries
11. Extract only the required kernel(s) from the built program
12. Pass all the arguments needed by the kernel to run to it one after the other
13. Enqueue the task by passing the command que and the kernel to the enqueue function
14. Read device memory object content back to the host memory
15. Use synchronization mechanism (e.g. *clFinish(commandqueue))* to ensure that kernel finished execution before next host function is run (if need be)

6.4.2 Accelerating circle detection task through OpenCL on GPU

In order to test the project's OpenCL based circle detection module on MATLAB, an OpenCL toolbox built by Radford Juang [56] was used. This toolbox helps create a gateway through which OpenCL kernels can be called from MATLAB, as MATLAB is currently only written to support CUDA. Prior to this, the circle detection function is written in C/C++/OpenCL and tested through Visual Studio on the AMD Radeon R5 graphics card to ensure functionality. The code except in code 6.0 below shows the OpenCL Kernel

Code 6.0 OpenCL Kernel Code for Circle Detection Using Circular Hough Transform

```
//OpenCL kernel for circle detection using Circular Hough Transform

//pragmas to enable floating point operations
#pragma OPENCL EXTENSION cl_khr_fp64 : enable
#pragma OPENCL EXTENSION cl_khr_global_int32_base_atomics : enable
#pragma OPENCL EXTENSION cl_khr_local_int32_base_atomics : enable

//Already created Sine and Cosine Look-up table for improved speed
__constant double cosineTableGlobal[360] = { 1, 0.540302, -0.416147, -
0.989992, -0.653644…}

__constant double sineTableGlobal[360] = { 0, 0.841471, 0.909297,
0.14112, -0.756802, -0.958924…}

__kernel void circularHough(
__global float *img,
__global int *accumulator,
__global int *xDimension,
__global int *yDimension,
__global int *range,
__global float *maxPixVal)
{
   //Define Variables
   int id = get_global_id(0); //Get Current Global ID
   int minRadius = (int) ceil(((double)((*yDimension)*0.1)));
   int radius=0;
   int a=0;
   int b=0;
   int pos = 0;

  //Get x,y,z index
   int idx = id;
   int zed = floor(((double)id) / (((double)(*xDimension)) *
((double)(*yDimension))));
   idx = id-(zed * (*xDimension) * (*yDimension));
   int wy = floor(((double)idx) / ((double)(*xDimension)));
   int ex = idx % (*xDimension);
   int val1 = (int) (wy + ((*xDimension)*ex));
   float val2= (float) (0.7*((float)(*maxPixVal)));
 if ((img[val1] > 0) && (img[val1] >= val2)) {
   for (int rad = 0; rad < (*range); rad++) { //Loop through radius range
     radius = minRadius + rad - 1;
     for (int tetha = 0; tetha < 360; tetha++) { //Every possible angle
        a = (int) ( ceil( ((double)ex) - (((double)radius) *
        ((double)cosineTableGlobal[tetha])) ));
        b = (int) ( ceil( ((double)wy) - (((double)radius) *
        ((double)sineTableGlobal[tetha])) ));
if (a > 0 && b > 0 && a <= (*xDimension) && b <= (*yDimension)) {
   pos = (int) ((rad * (*xDimension) * (*yDimension)) + (b *
     (*xDimension)) + a);
```

```
        atom_inc(&accumulator[pos]); //Increment accumulator content
    }
    }//end of theta loop
    }//end of radius loop
    }//end of if
    }
```

6.4.2.1 GPU Acceleration Performance

Figure 6.4 below shows the performance evaluation of the OpenCL version of the circle detection function, comparing it against its MATLAB and C++ versions. Recall that the circle detection function used for this analysis isn't the actual one used by the vision system, as that has already been highly optimized by MATLAB inc. The one benchmarked below is written based on the Circular Hough Transform theory and the main idea behind the benchmark is to show vividly that being able to pass graphics or computationally intensive operations to devices (such as GPUs and FPGAs) suited for them, we can tremendously improve the speed of od operation of our traffic sign detection and recognition vision system. Writing the OpenCL kernel for the MATLAB circle detection function ('imfindcircles') is recommended for future works.

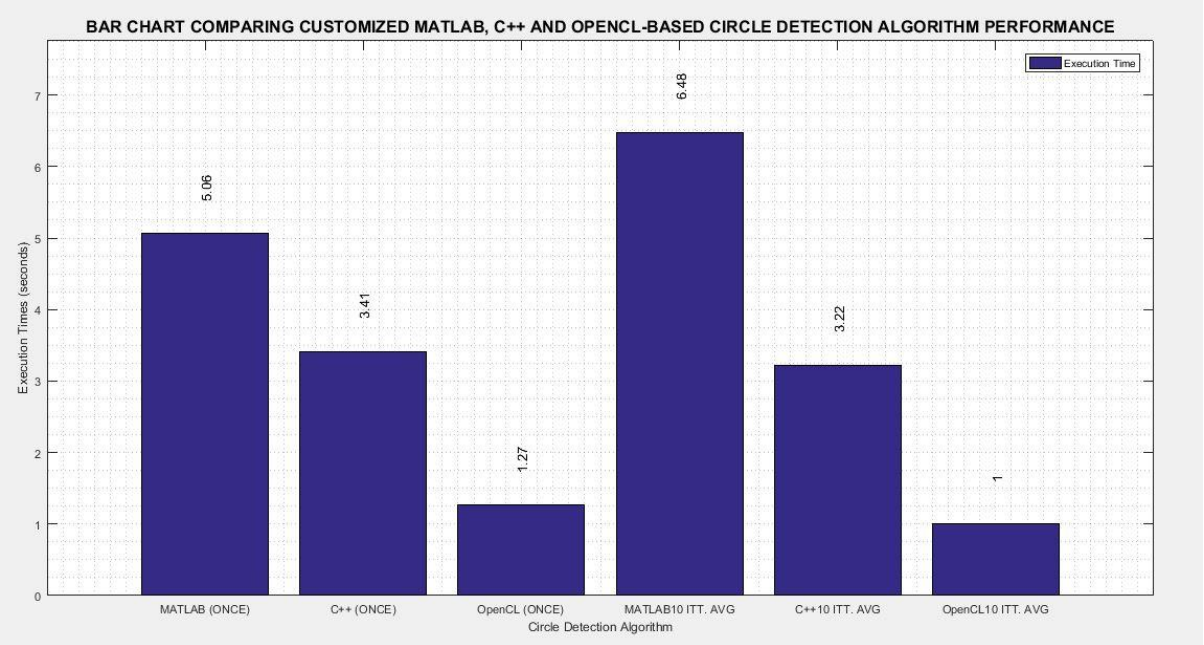

Figure 6.4 Figure Showing MATLAB vs C++ vs OpenCL Performance on Circle Detection (ITT = Itterations)

7.0 Discussion, Recommendations & Conclusion

Neural networks have now become successful architectures for modelling artificial intelligent systems in multiple scientific and engineering areas, including vision, speech recognition, natural language processing and the likes. However, ConvNets, a variation of the different deep learning architectures which utilize cascaded layers of nonlinear processing units for feature extraction are mostly used for image classification tasks.

In designing and implementing this computer vision system for traffic sign recognition and detection for autonomous vehicles, this project uses a particular model of the ConvNet family called the AlexNet. A model known to record very low error rates in image classification task involving millions of images to be classified into thousands of categories.

This project does not just stop at image/scene classification and detection on a normal CPU, but moved ahead to optimize the developed system so as to achieve higher rates of frame processing. It explores optimization mechanism such as vectorization, parallel thread dispatch, legacy programming language (C/C++) and even heterogeneous computing to ensure optimal performance. Real life video feeds are fed to the system (see Figure 6.2 and 5.11) and its performance is evaluated under these varying conditions.

7.1 Project results

In MATLAB, and by engaging different methods of optimization (both software and hardware) we have been able to push the performance of the system from about 1.5 seconds per frame (i.e. about half a frame per second) to between 25 to 30 frames per seconds using our C/C++ solution running across multiple cores. The pure MATLAB solution optimized through vectorization and multi core dispatch gives a maximum of just about 8 frames per second which is also way ahead of the un-optimized MATLAB version of the vision system. In a nutshell, the results achieved by this project are as listed below

1) *Design and implementation of a deep leaning model (AlexNet) for traffic sign classification which attained a classification accuracy of over 98%. See Figure 4.5 of section 4.4.2.*
2) *Design and implantation of multiple layers of traffic sign post detection mechanisms which attained a detection accuracy of over 99%. See Table 5.1 of section 5.3 and Figure 5.11 of section 5.3*
3) *Optimization of the detection algorithms performance from about half a frame per second to around 25 to 30 frames per second (classification operation included). This sums up to **over 50 time's improvement in speed** (C/C++ version). See Table 5.1 of section 5.3 and Figure 6.2 of section 6.3)*
4) *Practically proving the fact that further improvement in speed can be achieved through heterogeneous computing. i.e., by dedicating some parts of the computationally expensive detection functions to suitable devices such as GPUs and FPGA. See section 6.4.2.1*

7.2 Challenges faced & recommendation for future works

In order to do more on the heterogeneous computing part of this project, and be able to use my computer as the host platform dispatching the main .c file while the FPGA serve as an attached device, an high-end FPGA (such as the Stratix 10 GX FPGA) was required, as these are the device which are both OpenCL compatible and also provide allowance for PCIe

connectivity. It is quite costly to order these kinds of devices, as they range into thousands of US Dollars.

For future works, I suggest that the researcher focuses more on implementing the whole vision system in an heterogeneous computation architecture whereby different cores and compute devices take on different vision task e.g. CPU core 1 for classification, GPU for shape analysis, FPGA for circle detection etc., and by so doing be able to provide a vision system that can be relied on by very high speed vehicles. Note that this means that one of the high end FPGAs/SoC will have to be made available to the researcher from the onset of the project. Also, this means that the researcher now has to look into issues with energy consumption, as these devices are meant to be implemented in vehicles and should not be too energy consuming.

7.3 Skills Gained

During the course of this project, I have been able to gain quite a number of new skills. The under listed gives a summary of the key skills gained.

1) Knowledge and hands on with heterogeneous computing using OpenCL on GPUs and FPGAs
2) In-depth knowledge of Neural networks and its application in deep learning through hands on with Convolutional Neural Networks for traffic sign classification
3) In-depth Knowledge of Computer Vision. Understanding the theory behind various object detection concepts. Gained hands on with Sobel edge detection, Harris corner detection, Hough transform for shape analysis, connected component analysis etc.
4) More understanding of the need for code optimization in the development of computationally intensive applications.
5) More understanding of various hardware and software optimization techniques
6) Hands on with software testing and performance evaluation

7.4 Access to project

In order to be concise, only excerpts of project codes are put in this report. However, this project remains open source and is open to further contributions. For more details on the project resources and how to pull them, see Appendix B.1

Appendix A – More Details on Devices and Explanation of Concepts

A.1 More information on FPGAs

Field-Programmable Gate Arrays (FPGAs) are semiconductor devices consisting of a clusters of configurable logic blocks connected via programmable interconnects. These devices could be thought of as blank slates, as they are typically reprogrammable silicon chips. In other words, for FPGAs, there exist the possibility of programming the hardware itself, rather than just writing software which get to run on the predefined hardware. Now adays, high-end Field-Programmable Gate Arrays possess hundreds of thousands of re-configurable logic blocks, and also lots of already made functional units to aid the fast and efficient implementations of common logic functions on them. These logic blocks, fixed-function units and interconnect are programmed electronically by writing a configuration bit-stream into the device. As we know, they are essentially, a collection of logic cells called lookup tables (LUTs) surrounded by a series of interconnects, and these interconnects between the logic blocks can be regarded as a network of wire bundles running in-between the logic blocks, with switchboxes at each intersection. Considering the fact that the LUTs and interconnect are programmable, it therefore provides a flexible system that can implement almost any digital algorithm. It is important to note that its configuration is held in SRAM memory cells [57][58].

Application Specific ICs (ASICs) such as microcontrollers and microprocessors (standardized ASICs) on the other hand are fixed. They can be programmable but only at the software level. However, the interesting factor with FPGAs is the fact that they are programmable at the logic level, providing parallel, and faster implementations that a processor may not be able to equal.

FPGAs combine the best parts of different hardware systems. They are truly parallel, as different operations do not need to be executed sequentially, or compete for execution resources. This is on like the conventional processors we know which most times execute tasks sequentially. Each processing task is assigned a dedicated section of the chip for its operation and is not limited by the number of processor cores available

The main selling points of the FPGA technology are performance, time to market, cost, reliability and long-term Maintenance. Performance wise, the FPGA takes advantage of hardware parallelism, and easily exceeds the computing power of digital signal processors (DSPs) and ASICs by breaking the paradigm of sequential execution and accomplishing more per clock cycle. As mentioned earlier, FPGA chips are field-upgradable and do not require the time and expense involved with ASIC redesign. Replacing already deployed ASICs is usually not an easy task, however, reprogramming FPGAs already deployed is very easy.

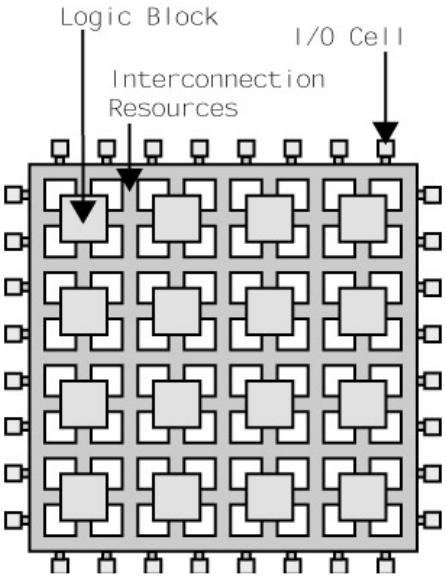

Figure A.0 Very High Level Representation of the FPGA Internal Structure

A.2 More information on GPUs

GPUs are computation devices specifically designed to be able to rapidly manipulate and alter memory locations, thereby improving their performances on some kinds of graphics computations, mostly in the area of image processing. GPUs are now being used embedded systems engineering to provide the extra edge needed by most graphics associated devices in terms of speed, as modern day GPUs are known to be very efficient in the manipulation of graphics data. Their highly parallel architecture enables them do better with tasks which require processing of large blocks of data in parallel. This kind of characteristic is not usually achievable with CPUs as they have only few cache layers which may not be as closely knitted to the cores, as we have in GPUs.

A.3 OpenCL operation architecture

In the OpenCL architecture, there exist one CPU-based ‘Host’ which controls multiple ‘Compute Devices’ such as CPUs, DSP chips, GPUS etc. Each of these compute devices also consists of multiple ‘Compute Units’ such as Arithmetic and Logic Units (ALUs), processing unit groups on multicore CPUs. And then finally, each of these Compute Units have multiple ‘Processing Elements’ upon which the OpenCL kernels are executed.

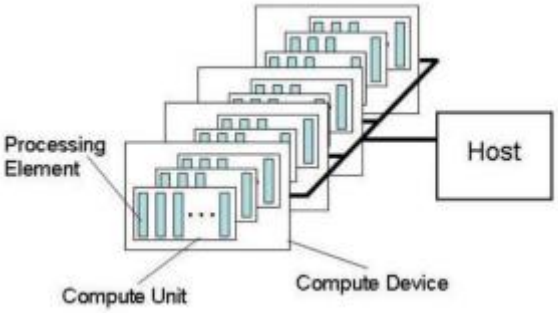

Figure A.1 OpenCL Operation Architecture [59]

An OpenCL application is split into two parts, the host program and the kernel(s). The host program is executed on the CPU of the system, and can perform any functions or computations as if it were a regular C program. In addition, an OpenCL host program is able

to launch one or more kernels in order to speed up computation. A kernel is a special function written in OpenCL C that performs some user-defined computation.

A.4 OpenCL execution model

The OpenCL host uses the OpenCL API to query and select desired compute devices, submit work to these devices and manage the workload across compute contexts and work-queues [60]. However, looking deeper into the execution of these work/tasks, there exists OpenCL kernels running in parallel on the various PEs on the compute device over a predefined N-dimensional computation domain. Each execution is termed a work-item, and these work-items are grouped together into 'work-groups/thread blocks'. In summary, the execution model is an internetwork of 'fine grained data parallelism and thread parallelism, nested within coarse-grained data parallelism and task parallelism' [61]. The host primarily creates 'Context' to manage OpenCL resources and shares common memory buffers known as the memory object with the compute device. The command Queue manages the execution of kernels in and out of order, although commands are queued in order. The command queue accepts kernel execution commands, memory management commands and synchronization commands.

A.5 OpenCL actual execution pipeline

The simplified model of the OpenCL execution pipeline is as listed below

1. CPU host defines an N-dimensional computation domain over some region of DRAM memory.
2. Every spot on the array will be a work-item. Each work-item is slated to execute the same Kernel concurrently within the compute unit
3. Host groups the work items into work groups. The work items within the work group share the same memory.
4. These work-groups are placed onto a work-queue.
5. DRAM memory is then loaded into the global memory and each work group on the compute device is executed.

Note that each PE executes purely sequential code.

A.6 OpenCL memory architecture

OpenCL defines a four-level memory hierarchy for the compute device [62] which are the

1. Global Memory
2. Per-element Private Memory (registers).
3. Read-only Memory
4. Local Memory

The global memory is shared by all processing elements, but has high access latency while the local memories are shared between a group of processing elements. The read only memory however can be written to by the host CPU but not the compute device itself. It is important to note that devices may or may not share memory with the host CPU [62]. The lowest level execution unit has a small private memory space for program registers.

The memory architecture is structured in such a way that each work group has an independent memory access but can communicate with other work groups through shared memory (Global Memory).

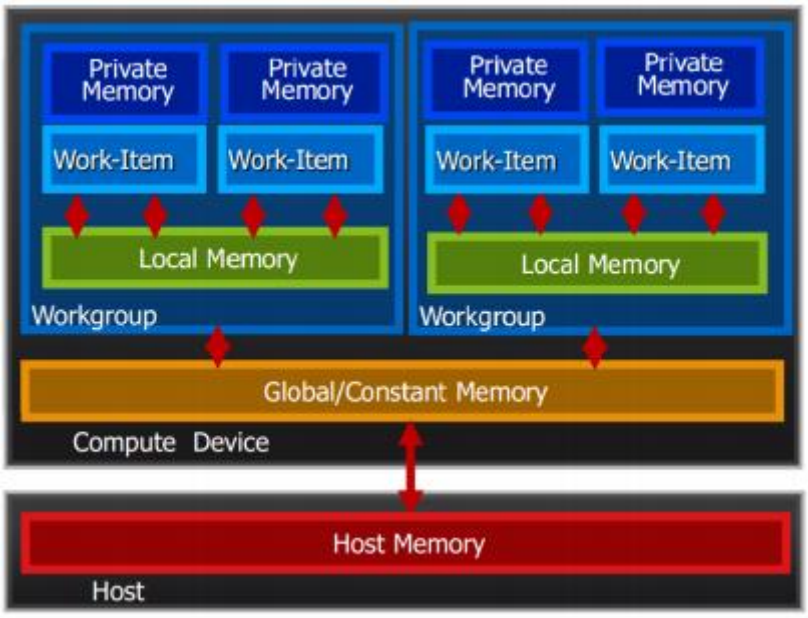

Figure A.2 OpenCL Memory Architecture [59]

This model allows work items to be able to synchronize with each other, as they share the same local memory and also work groups can communicate with each other because they share the same global memory. It is important to check the return conditions from the host API when statically allocating local data (per work-group). This will help detect instances of allocating too much per work-group.

A.7 OpenCL programming language structure

The OpenCl programming language is based on C. Pointers are represented using qualifiers such as '__global', '__constant', '__local' and '__private' and its functions are represented with the entry statement '__kernel'. Recursions, variable-length array, function pointers and bit fields are not allowed. It provides fixed-length vector types such as float4 which is designed to map onto SIMD instructions sets. Float4s are simply 4-vector of single-precision floats.

A.8 Limitations in OpenCL

In OpenCL, global work size must be a multiple of the work-group size and work-group size must be less than or equal to the maximum allowable kernel work group size on the device. The maximum work-group size indicates the maximum concurrent threads within a work group on the device.

A.9 Types of parallel programming

There exist two major types of parallel programming which are:

- Data-parallel programming: Data is distributed across different nodes, which operate on them in parallel. Execution in these nodes are deemed independent.
- Task Parallel Programming: Distributed tasks/codes concurrently performed/acted upon by processes or threads across different processors

OpenCL API gives room for both of this form of parallelism, however, it is important to consider how well the kind of parallelism we want to implement compliments the OpenCL memory architecture e.g. access to global memory is slow, therefore, one can write codes whereby work-groups are separated by task parallel programming (since threads within a work-group can share local memory) and work items by data parallel programming. At times,

one instruction executes identical code in parallel by different threads and each thread executes the code with different data.

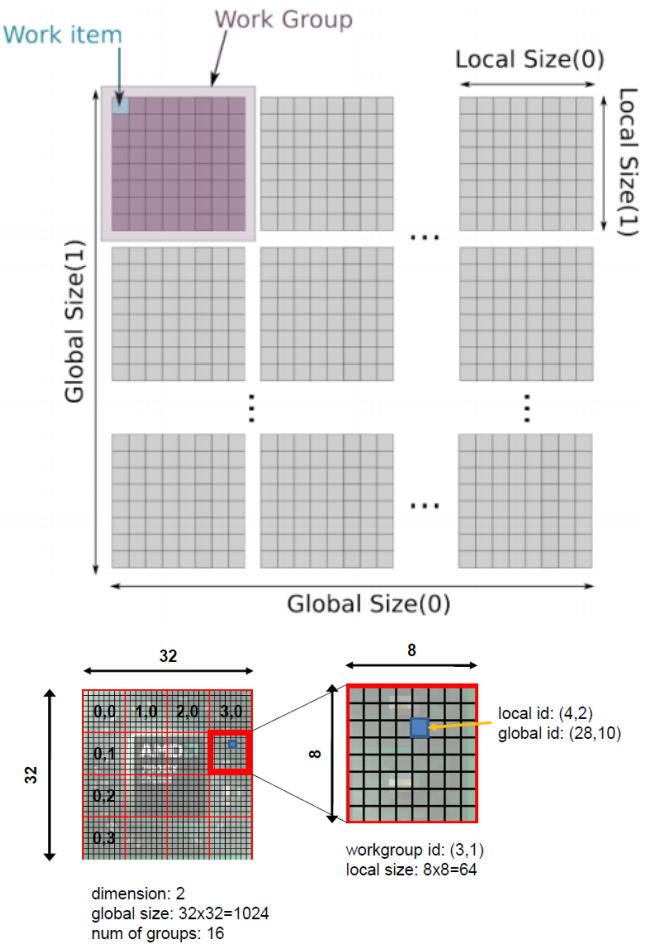

Figure A.3 Figure Showing 2D Data-Parallel execution in OpenCL [63]

Appendix B – Code Breakdown and Excerpts

B.1 Code details and distribution

This section gives a breakdown of the main codes used in this project and gives a brief explanation on the task(s) performed by each of the function.

	MATLAB Codes	C/C++ Codes	OpenCL Codes
CLASSIFICATION			
1	*AlexNetNN.m*	-	-
2	*callAlex.m*	-	-
3	*classifyImg.m*	-	-
4	*createIMDB.m*	-	-
5	*sceneClassifier.m*	-	-
DETECTION MODULE			
1	*analyseCriticalAreas.m*	*analyseCriticalAreas.cpp*	-
2	*CCA.m*	*CCA.cpp*	-
3	*detectCircle.m*	*detectCircle.cpp*	-
4	*detectCorner.m*	*detectCorner.cpp*	-
5	*detectEdge.m*	*detectEdge.cpp*	-
6	*getROI.m*	-	-
7	*myDetectCircle.m*	*CircleDetection.cpp* *VarFile.cpp* *setUpBase.cpp*	Kernel.cl
8	*nonParallelDetection.m*	-	-
9	*parallelDetection.m*	-	-
10	*shapeAnalyser.m*	shapeAnalyser.cpp	-
SYSTEM EVALUATOR			
1	*evaluateDetectionSystem.m*	-	-
2	*evaluateOpenCLPerformance.m*	-	-
3	*multiEpochAnalyser.m*	-	-
4	*networkTest.m*	-	-
5	*testDetection.m*	-	-
TEST BENCH AND VISUAL VALIDATOR			
1	*cornerTestBench.m*	-	-
2	*detectCircleTestBench.m*	-	-
3	*drawDetectedCircle*	-	-
4	*passVideoToVision.m*	-	-
5	*viewCircularHough*	-	-
OTHERS			
1	*setGlobalVAriables.m*	-	-
2	*getGlobalVariables.m*	-	-
3	*getGrayScale.m*	-	-
4	*MapRegion.m*	-	-

The list below gives a short explanation on each of the function.

- **AlexNetNN.m:** Used to create alexnet model, link to the IMDB and train
- **callAlex.m:** Used to call AlexNetNN and also set some post training parameters

- **classifyImg.m:** Used to run classification on the detected ROI
- **createIMDB.m:** Used to create an image database for the NN training
- **sceneClassifier.m:** Used to validate input image to system, set detection algorithm parameters and call parallel/nonparallelDetection
- **CCA.m:** Used to perform connected component analysis on input image
- **detectCircle.m:** For circle detection
- **detectCorner.m:** For corner detection (Harris method)
- **detectEdge.m:** For edge detection (Sobel's method)
- **getROI.m:** Used to call CCA and also set some pre-CCA call conditions
- **myDetectCircle.m:** Customized circle detection algorithm (Circular Hough Transform)
- **networkTest.m:** Used to test individual CNN epoch outputs
- **testDetection.m:** Used to test individual detection functions
- **getGrayScale.m:** To get image grayscale
- **MapRegion.m:** Region class used for CCA
- **viewCircularHough:** Used to have a look into how the circular hough transform operates
- **shapeAnalyser.m:** Analyses and searches for triangles, rectangles, octagon and diamond shapes

- **cornerTestBench.m:** Testbench for corner detection function
- **setGlobalVariables.m:** Used to set global variables
- **getGlobalVariables.m:** Used to get global variables
- **drawDetectedCircle.m:** Takes in image, circle radius and centre co-ordinates and produces the circle on top of the received imge
- **passVideoToVision.m:** To feed video stream to the vision system, so as to evaluate
- **parallelDetection.m:** Used to execute all detection algorithms in a parallel way
- **multiEpochAnalyser.m:** Used to test a wide range of CNN epoch outputs so as to detect overfitting
- **detectCircleTestBench.m:** Test bench for circle detection function performance
- **analyseCriticalAreas.m:** Used to analyse critical areas as explained in 5.2.2
- **nonParallelDetection.m:** Used to execute all detection algorithms in a non-parallel way

- **evaluateDetectionSystem.m:** Used to run evaluation on all the individual detection functions
- **evaluateOpenCLPerformance.m:** Used to evaluate the performance of the OpenCL based parts of the vision system application

B.1.1 Code distribution

To get access to the codes used in this project, visit
https://github.com/OluwoleOyetoke?tab=repositories

B.2 AlexNet training and creation code

Code B.0 Excerpt of Code Used to Create and Train the AlexNet Model (SimpleNN)

```
function [net  trainingInfo] = AlexNetNN( imageDB, dumpLocation );
%AlexNetNN: Used to Create AlexNet and Train it based on specified
%imageDB
%   Creates A SimpleNN AlexNet Network, initializes its weight values and
%   trians it based on the suplied ImageMDB over several Epochs
%{
            Created on: 31st March, 2017
             Author: Oluwole Oyetoke Jnr
             Using MATLAB 2016

•   Every fully-connected Layer has 4096 neurons
•   Max-pooling layers follow first, second, and fifth convolutional
layers
•   Response-normalization layers follow the first and second
convolutional layers
•   The ReLU non-linearity is applied to the output of every
convolutional and fully-connected layer
%}

% Install and compile MatConvNet Library (needed only once).
untar('http://www.vlfeat.org/matconvnet/download/matconvnet-1.0-
beta23.tar.gz') ;
cd matconvnet-1.0-beta23
run matlab/vl_compilenn;

% Setup MatConvnet.
run matlab/vl_setupnn;

%Choose SimpleNN network type. There also exists the Directed Acyclic
Graph
%(DagNN). An object-oriented wrapper for CNN with complex topologies
opts.networkType = 'simplenn' ;

%Switch away from MATLab lagacy random number generator method (caution)
rng('default');
rng(0);

%create empty layer struct where the AlexNet network will be modeled into
net.layers = {} ;
networkInputSize= [227 227 3];

%Validation
if (nargin ~= 2)
 error('AlexNetNN:Input_Argument_Error','This function works with 2 input
arguments -imageDB, dumpLocation- ');
end

%BEGINING OF NETWORK DEFINITION

%FIRST CONVOLUTIONAL BLOCK
%The first convolutional layer filters the 227×227×3 input image with
%96 kernels of size 11×11×3 with a stride of 4 pixels. Bias of 1.
net.layers{end+1} = struct('type', 'conv', ...
    'weights', {initializeWeights(11,11,3,96)}, ...
    'stride', 4, ...
    'pad', 0, ...
```

```
    'name', 'conv1') ;
net.layers{end+1} = struct('type', 'relu', 'name', 'relu1') ;
net.layers{end+1} = struct('type', 'lrn', 'name', 'lrn1') ;
net.layers{end+1} = struct('name', 'pool1_cv_layer1', ...
    'type', 'pool', ...
    'method', 'max', ...
    'pool', [3 3], ...
    'stride', 2, ...
    'pad', 0) ;

%SECOND CONVOLUTIONAL BLOCK
%Divide the 96 channel blob input from block one into 48 and process
%independently
net.layers{end+1} = struct('type', 'conv', ...
    'weights', {initializeWeights(5,5,48,256)}, ...
    'stride', 1, ...
    'pad', 2, ...
    'name', 'conv2') ;
net.layers{end+1} = struct('type', 'relu', 'name', 'relu2') ;
net.layers{end+1} = struct('type', 'lrn', 'name', 'lrn12') ;
net.layers{end+1} = struct('name', 'pool2_cv_layer2', ...
    'type', 'pool', ...
    'method', 'max', ...
    'pool', [3 3], ...
    'stride', 2, ...
    'pad', 0) ;

%THIRD, FOURTH AND FIFTH CONVOLITIONAL LAYER
%The third, fourth, and fifth convolutional layers are connected to one
%another without any intervening pooling or normalization layers.

%THIRD BLOCK
%The third convolutional layer has 384 kernels of size 3 × 3 × 256
%connected to the (normalized, pooled) outputs of the second
convolutional layer
net.layers{end+1} = struct('type', 'conv', ...
    'weights', {initializeWeights(3,3,256,384)}, ...
    'stride', 1, ...
    'pad', 1, ...
    'name', 'conv3') ;
net.layers{end+1} = struct('type', 'relu', 'name', 'relu3') ;

%FOURTH BLOCK
%The fourth convolutional layer has 384 kernels of size 3 × 3 × 192
net.layers{end+1} = struct('type', 'conv', ...
    'weights', {initializeWeights(3,3,192,384)}, ...
    'stride', 1, ...
    'pad', 1, ...
    'name', 'conv4') ;
net.layers{end+1} = struct('type', 'relu', 'name', 'relu4') ;
%FIFTH BLOCK
%the fifth convolutional layer has 256 kernels of size 3 × 3 × 192
net.layers{end+1} = struct('type', 'conv', ...
    'weights', {initializeWeights(3,3,192,256)}, ...
    'stride', 1, ...
    'pad', 1, ...……..
```

B.3 IMDB creation code

Code B.1 Code Used to Create the IMDB 'Struct' from the GTSRB Dataset

```
function [] = createIMDB(trainingImageLocation, dumpSpace)
%createIMDB: Function used to create IMDB
%   Function scans through the real image database on the computer and
%   creates the IMDB of well labelled and classified images. Meta data
such
%   as image categorizations are also included in the created db to guide
%   any user who might need to use the IMDB in the future
%   70 percent of the data is used for training
%   20 percent for validation
%   10 percent reserved for testing
%   Training Set  : to fit the parameters [i.e., weights]
%   Validation Set: to tune the parameters [i.e., architecture]
%   Testset       : to assess the performance [i.e. predictive power]



%{
   Created on: 31st March, 2017
   Author: Oluwole Oyetoke Jnr
   Using MATLAB 2016
%}

%Validation
if nargin ~= 2
    error('createIMDB:Input_Argument_Error','This function works with 2
input argument -trainingImageLocation, dumpSpace- ')
end

%IMDB Creation Start Time
datenow = datetime('now','Format','dd-MMM-yyyy HH:mm:ss');
fprintf('Start Time: %s\n\n',datenow);

%Create an empty IMDB structure
imdb = struct();
categories = {'speed_20', 'speed_30','speed_50','speed_60','speed_70',...
    'speed_80','speed_less_80','speed_100','speed_120',...
    'no_car_overtaking','no_truck_overtaking','priority_road',...
    'priority_road_2','yield_right_of_way','stop','road_closed',...
    'maximum_weight_allowed','entry_prohibited','danger','curve_left',...
    'curve_right','double_curve_right','rough_road','slippery_road',...
    'road_narrows_right','work_in_progress','traffic_light_ahead',...
    'pedestrian_crosswalk','children_area','bicycle_crossing',...
    'beware_of_ice','wild_animal_crossing','end_of_restriction',...
    'must_turn_right','must_turn_left','must_go_straight',...
    'must_go_straight_or_right','must_go_straight_or_left',...
    'mandatroy_direction_bypass_obstacle',...
    'mandatroy_direction_bypass_obstacle2', ...
    'traffic_circle','end_of_no_car_overtaking',...
    'end_of_no_truck_overtaking'};

datasets = {'train', 'validate', 'test'};

%To Create an IMDB scaled to a different size, simply change netInputSize
netInputSize = [32 32];

%.ppm (portable pixmap format) is used in this project
primaryTrainingDataPath = trainingImageLocation;
```

```
try
    %Loads all the content of the  training folder
    trainingFolderStruct = dir([primaryTrainingDataPath]);
catch
    error('createIMDB2:Traing_Image_Location_Error','Error Encounterd
When Loading Image Data From Folder')
end
[noOfTrainingFolders d] = size(trainingFolderStruct);
dirFlags = [trainingFolderStruct.isdir];
subFolderList = trainingFolderStruct(dirFlags);

%Loop through the training image folder to get total number of images in
DB
%Main folder contains subfolders of images for each training class
imageCount=0;
for mainLoopCount = 3:noOfTrainingFolders
    secondaryTrainingDataPath = fullfile(primaryTrainingDataPath,...
        subFolderList(mainLoopCount).name, '*.ppm');
    subFolderStruct = dir([secondaryTrainingDataPath]);
    noOfContents2= numel(subFolderStruct);
    imageCount =imageCount+noOfContents2;
    % fprintf('Number of Images in Training Class %s = %d\n', ...
    %subFolderList(mainLoopCount).name, noOfContents2);
end
fprintf('Number of Training Images in Total: %d\n', imageCount);


%Initialize part of the imdb structure
imdb.meta.sets = {'train', 'validate', 'test'};

%Possible Image Categories
imdb.meta.categories = categories;

%AlexNet Uses 227 by 227 by 3 images
imdb.images.data = ones(netInputSize(1), netInputSize(2), 3, imageCount,
'single');
imdb.images.labels = ones(1,imageCount, 'single'); %Image label
% vector indicating to which set an image belong,
%i.e., % training, validation, test etc.
imdb.images.set = ones(1, imageCount, 'uint8');

fprintf('Each image will be resized to %d by %d by 3 \n',
netInputSize(1), netInputSize(2));

%Loop through Dataset, appropriately dimension all contents, label,
%classify and place in sets
imageCounter=1;
for mainLoopCount = 3:noOfTrainingFolders

    actualPos = mainLoopCount-2;
    toWorkOn = char (categories(actualPos));
    fprintf('%d. Loading and working on training, validation and test
images for '' %s '' traffic sign\n',actualPos, toWorkOn);……
```

B.4 Multi-epoch testing code

Code B.2 Multi Epoch Analyser Code Excerpt

```
function [performanceArray] = multiEpochAnalyser( imdb,
epochOutNetLocation )
%multiNetworkAnalyser: This function analyses multiple Epochs of a
network
%based on test data in the IMDB to determine the point where overfitting
%sets in
%   This function takes in the address of a file location where by
multiple
%   epoch outputs of a network is saved over the training process. Test
%   analysis is run through each of the netwok structures to determine
the
%   point where training most likely must have started leading to
%   overfitting. With this system designer can afford to train their
%   network over multiple epochs and just run this function to determine
%   the most optimal of the different generated networks after each stage
%   of the training process. Note that the 'epochOutNetLocation' should
%   only contain the saved network structure after the various epochs of
%   the training process.

%{
   Created on: 1st April, 2017
   Author: Oluwole Oyetoke Jnr
   Using MATLAB 2016
%}



%Validation
if nargin ~= 2
    error('multiEpochAnalyser:Input_Argument_Error','This function works
with 2 input argument -imdb, epochOutNetLocation- ')
end

%Analysis Start Time
datenow = datetime('now','Format','dd-MMM-yyyy HH:mm:ss');
fprintf('Start Time: %s\n\n',datenow);



%Check the IMDB meta to know set assigned for testing
sets = imdb.meta.sets;
row=0;
col=0;
[row,col]=find(strcmp(sets,'test'));
if(col==0)
    error('networkTest:Sets_Error','There are no designated image in imdb
for testing of network');
end
testImgSet = col;



%Make sure your files are arranged in ascending numerical order e.g 01,
%02.....09, 10, 11.....
epochOutPath = fullfile(epochOutNetLocation, '*mat');

try
```

```
    %Loads all the content of the  epoch out folder
    epochOutStruct = dir([epochOutPath]);
catch
    error('multiEpochAnalyser:Epoch_Out_Location_Error','Error Encounterd
When Loading Epoch out Networks From Folder')
end
noOfNetworks = length(epochOutStruct);

 %Automated Testing. Get No. of test images available in DB
    [testImgPositions] = find(imdb.images.set == testImgSet);
    performanceArray = zeros(noOfNetworks,1);

    noOfTestCases=0;

fprintf('Multi Epoch Analysis in Progress.....\n');
%Pick each network and perform performance test on it
for loopCount = 1:noOfNetworks
  performance=0;
    pass=0;
     noOfTestCases=0;
    %Pick Network
    netLocation =    fullfile(epochOutNetLocation,
epochOutStruct(loopCount).name);
 network = load(netLocation);

 %Perform Test
    for innerLoopCount = 1:length(testImgPositions)
        %Scan through DB and use test images to evaluate network
        testImg =
imdb.images.data(:,:,:,testImgPositions(innerLoopCount));
        testImgCategory =
imdb.images.labels(1,testImgPositions(innerLoopCount));
        [prediction, score] = miniClassifierFunction(testImg,
network.net);
        noOfTestCases = noOfTestCases+1;
        if(prediction==testImgCategory)
            pass=pass+1;
        end

    end %end of inner loop
     %(Calculate Network Performance)
          performance = ((pass*100)/length(testImgPositions));
          networkName = epochOutStruct(loopCount).name;

          performanceArray(loopCount) = performance;
          percentageCompletion = uint8 ((loopCount*100)/noOfNetworks);
          fprintf('''%s'' Performance: %.3f%% Accuracy\n%d%% of process
completed so far\n\n',networkName, performance, percentageCompletion);
end %end of loop accross all netowrk


%Plot performance Histogram
% Add title and axis labels. Create Label Cell arrayAdd a text string
above each bin
labels = {epochOutStruct(1).name};
for i = 2:length(performanceArray)
    labels(end+1) = {epochOutStruct(i).name};
end
barr = bar(performanceArray, 'g');…….
```

B.5 Connected Component Analysis C++ Code

Code B.3 Connected Component Analysis MEX Code

```
#include "mex.h"
#include <stdlib.h>
#include <cstdlib>
#include <Math.h>
#include <stdio.h>
#define xDim 500
#define yDim 500
#define zDim 3
#define windowSize 20
#define stepSize 10
#define beacon ceil((windowSize*windowSize) / 20)
struct Region { //Create region struct
int region; //region unattended to
int value; //default value
int xCoord; //default value
int yCoord; //default value
struct Region* left; //empty MapRegion object
struct Region* right; //empty MapRegion object
struct Region* top; //empty MapRegion object
struct Region* bottom; //empty MapRegion object
public:
//initialize variables
Region() : region (-1), value(-1), xCoord(-1), yCoord(-1) {}
};
class MapRegion {
public:
MapRegion() {};
~MapRegion() {};
};
void CCAmex(int* inputArray, int* outputArray);
int getLocation(int x, int y, int width);
void CCAmex(int* inputArray, int* outputArray) {
//Initialize aprameters and variables
int startX = 0; int startY = 0; int stopX = windowSize - 1; int stopY =
windowSize - 1;
int xCounter = 0;
int yCounter = 0;
int regionSum = 0; //Sum of the number of Pixels which are potentialy
members of the ROI
const int xBlocks = ((xDim - windowSize) / stepSize) + 1;
const int yBlocks = ((yDim - windowSize) / stepSize) + 1;
const int totalBlocks = xBlocks * yBlocks;
struct Region* Map[totalBlocks];
int **regionShedd = new int*[xDim];
int* regionSizeTracker = (int*)calloc(totalBlocks, sizeof(int));
int* regionMap = (int*)calloc(totalBlocks, sizeof(int));
int regionCounter = 0;
int location = 0; int left = 0; int right = 0;int top = 0;int bottom = 0;
int* myLeftNeighRegion = NULL;
int* myRightNeighRegion = NULL;
int* myTopNeighRegion = NULL;
int* myBottomNeighRegion = NULL;
int cnt=0;
int xCount = 0; int yCount = 0;
/* The gateway function */
/*
* nlhs - Number of output (left-side) arguments, or the size of the plhs
array.
```

```
* plhs - Array of output arguments.
* nrhs - Number of input (right-side) arguments, or the size of the prhs
array.
* prhs - Array of input arguments.
*CALL MEX FUNCTION THIS WAY IN MATLAB CODE
transpose = img';
img1D = transpose(:)'; //TURNING IMAGE TO 1D ARRAY
[output] = CCAmex(uint32(img1D))
*GET OUTPUT THIS WAY
x = output(1);
y = output(2);
width = output(3);
height = output(4);
*/
void mexFunction( int nlhs, mxArray *plhs[],int nrhs, const mxArray
*prhs[])
{
int *inMatrix; /* 1xN input matrix */
int *outMatrix; /* output matrix */
int outputSize=4;
const int xBlocks = ((xDim - windowSize) / stepSize) + 1;
const int yBlocks = ((yDim - windowSize) / stepSize) + 1;
const int totalBlocks = xBlocks * yBlocks;
/* To check that there is only one input argument i.e inMatrix */
if(nrhs!=1) {
mexErrMsgIdAndTxt("MyToolbox:arrayProduct:nrhs","One inputs required.");
}
/* To check that there is only one output argument i.e outMatrix */
if(nlhs!=1) {
mexErrMsgIdAndTxt("MyToolbox:arrayProduct:nlhs","One output required.");
}
/* make sure the first input argument is not a scalar i.e it is a matrix
*/
if(mxGetNumberOfElements(prhs[0])<=1) {
mexErrMsgIdAndTxt("MyToolbox:arrayProduct:notScalar","Input multiplier
must be a 1D
matrix.");
}
/* make sure the first input argument is type double or uint8 */
if( !mxIsDouble(prhs[0]) && !mxIsUint8(prhs[0]) && !mxIsUint32) {
mexErrMsgIdAndTxt("MyToolbox:arrayProduct:notDouble","Input matrix must
be type
double or uint8.");
}
/* check that number of rows in first input argument is 1 */
if(mxGetM(prhs[0])!=1) {
mexErrMsgIdAndTxt("MyToolbox:arrayProduct:notRowVector","Input must be a
row
vector.");
}
/* create a pointer to the real data in the input matrix */
inMatrix = (int32_T *) mxGetPr(prhs[0]);
/* create the output matrix */
plhs[0] =
mxCreateNumericMatrix(1,(mwSize)outputSize,mxINT32_CLASS,mxREAL);
/* get a pointer to the real data in the output matrix */
outMatrix = (int32_T *) mxGetPr(plhs[0]);
/* call the computational routine */
CCAmex(inMatrix,outMatrix);
return;
}
```

References

[1] Vincent Cheung, Kevin Cannons. An Introduction to Neural Networks, Signal & Data Compression Laboratory Electrical & Computer Engineering University of Manitoba Winnipeg, Manitoba, Canada

[2] Luis von Ahn, Manuel Blum, Nick J. Hopper, and John Langford. CAPTCHA: Telling humans and computers apart. In Lecture Notes in Computer Science, pages 294–311. Springer

[3] "A Brief Introduction to Neural Networks [D. Kriesel]", Dkriesel.com, 2017. [Online]. Available: http://www.dkriesel.com/en/science/neural_networks. [Accessed: 17- Apr- 2017].

[4] David Warmflash, MD “Santiago Ramón y Cajal and Camillo Golgi” Vision learning Vol. SCIRE-2 (8), 2016.

[5] Rick McKeon, Neural Networks, a non-technical introduction, A Fascinating Look inside this technology , 2016

[6] Fundamentals of neural networks: architectures, algorithms, and applications. Prentice-Hall, Inc. Upper Saddle River, NJ, USA, Copyright 1994

[7] "Artificial neuron", En.wikipedia.org, 2017. [Online]. Available: https://en.wikipedia.org/wiki/Artificial_neuron. [Accessed: 11- Mar- 2017].

[8] A.D.Dongare, R.R.Kharde, Amit D.Kachare, “Introduction to Artificial Neural Network”, International Journal of Engineering and Innovative Technology (IJEIT) Volume 2, Issue 1, pp 189 – 194, July 2012,

[9] John A. Bullinaria, Biological Neurons and Neural Networks, Artificial Neurons, Neural Computation : Lecture 2 – University of Birmingham, 2015

[10] Ciro Donalek. (April 2011), Supervised and Unsupervised Learning, Carlifornia Institute of Technology

[11] A. Atiya, "An unsupervised learning technique for artificial neural networks", Neural Networks, vol. 3, no. 6, pp. 707-711, 1990.

[12] G. Carpenter and S. Grossberg, "The ART of adaptive pattern recognition by a self-organizing neural network", Computer, vol. 21, no. 3, pp. 77-88, 1988.

[13] Prof. Raul Rojas, Neural networks, 1st ed. Berlin: Springer-Verlag, 1996.

[14] "Rohan & Lenny: Neural Networks & The Backpropagation Algorithm, Explained", A Year of Artificial Intelligence, 2017. [Online]. Available: https://ayearofai.com/rohan-lenny-1-neural-networks-the-backpropagation-algorithm-explained-abf4609d4f9d#.dvlrb31gu. [Accessed: 11- Mar- 2017].

[15] Mirza Cilimkovic, ‘Neural Networks and Back Propagation Algorithm’, Institute of Technology Blanchardstown, Blanchardstown Road North, Dublin, Ireland, 2015

[16] "Cost Functions in a Neural Network - Machine Philosopher", Machine Philosopher, 2017. [Online]. Available: http://www.machinephilosopher.com/cost-function-neural-network-intro/. [Accessed: 11- Mar- 2017].

[17] "A Step by Step Backpropagation Example", Matt Mazur, 2017. [Online]. Available: https://mattmazur.com/2015/03/17/a-step-by-step-backpropagation-example/. [Accessed: 11- Mar- 2017].

[18] 5 algorithms to train a neural network | Neural Designer", Neuraldesigner.com, 2017. [Online]. Available: https://www.neuraldesigner.com/blog/5_algorithms_to_train_a_neural_network. [Accessed: 11- Mar- 2017].

[19] "Neural Network Basics", Webpages.ttu.edu, 2017. [Online]. Available: http://www.webpages.ttu.edu/dleverin/neural_network/neural_networks.html. [Accessed: 11- Mar- 2017].

[20] "Machine Learning FAQ", Sebastian Raschka's Website, 2017. [Online]. Available: https://sebastianraschka.com/faq/docs/closed-form-vs-gd.html. [Accessed: 11- Mar- 2017].

[21] C. Naugler, E. Mohammed, M. Mohamed and B. Far, "Peripheral blood smear image analysis: A comprehensive review", Journal of Pathology Informatics, vol. 5, no. 1, p. 9, 2014.

[22] Lawrence, S., Giles, C.L., and Tsoi, A.C. (1996), "What size neural network gives optimal generalization? Convergence properties of backpropagation". Technical Report UMIACS-TR-96-22 and CS-TR-3617, Institute for Advanced Computer Studies, University of Maryland, College Park

[23] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. Backpropagation applied to handwritten zip code recognition. Neural Computation, 1(4):541-551, 1989

[24] Prof. Dr. Bastian Leibe, David Stutz, Lucas Beyer, “Understanding Convolutional Neural Networks”, Computer Vision Seminar Report, RWTH Aachen University, August 30th, 2014

[25] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Parallel distributed processing: Explorations in the microstructure of cognition. chapter Learning Representations by BackPropagating Errors, pages 318–362. MIT Press, Cambridge, 1986.

[26] Keiron Teilo O'Shea, ‘An Introduction to Convolutional Neural Networks’, Research Gate’, 15th December, 2015

[27] Fei Fei Li, Justin honson, Andrej Karpathy, Convolutional Neural Networks for Visual Recognition, CS231 Lecture Note, Stanford University, United States

[28] "An Intuitive Explanation of Convolutional Neural Networks", the data science blog, 2017. [Online]. Available: https://ujjwalkarn.me/2016/08/11/intuitive-explanation-convnets/. [Accessed: 17- Mar- 2017].

[29] Scherer, Dominik; Müller, Andreas C.; Behnke, Sven (2010). "Evaluation of Pooling Operations in Convolutional Architectures for Object Recognition" (PDF). Artificial Neural Networks (ICANN), 20th International Conference on. Thessaloniki, Greece: Springer. pp. 92–101.

[30] Andrea Vedaldi Karel Lenc Ankush Gupta, MatConvNet Convolutional Neural Networks for MATLAB, 2016

[31] P. Joshi, "What Is Local Response Normalization In Convolutional Neural Networks", PERPETUAL ENIGMA, 2017. [Online]. Available: https://prateekvjoshi.com/2016/04/05/what-is-local-response-normalization-in-convolutional-neural-networks/. [Accessed: 20- Mar- 2017].

[32] Krizhevsky, A., I. Sutskever, and G. E. Hinton. "ImageNet Classification with Deep Convolutional Neural Networks. " Advances in Neural Information Processing Systems. Vol 25, 2012.

[33] "MatConvNet", Vlfeat.org, 2017. [Online]. Available: http://www.vlfeat.org/matconvnet/. [Accessed: 05- Apr- 2017].

[34] "German Traffic Sign Benchmarks", Benchmark.ini.rub.de, 2017. [Online]. Available: http://benchmark.ini.rub.de/?section=gtsrb&subsection=dataset. [Accessed: 12- Apr- 2017].

[35] Dan Hendrycks, Kevin Gimpel, Adjusting For Dropout Variance In Batch Normalization And Weight Initialization, arXiv:1607.02488v2, 23 Mar 2017

[36] "Army in Europe Pamphlet 190-34", Usareurpracticetest.com, 2017. [Online]. Available: http://www.usareurpracticetest.com/germany/documents/AEP190-34.htm. [Accessed: 12- Apr- 2017].

[37] "Momentum and Learning Rate Adaptation", Willamette.edu, 2017. [Online]. Available: https://www.willamette.edu/~gorr/classes/cs449/momrate.html. [Accessed: 12- Apr- 2017].

[38] "Metacademy", Metacademy.org, 2017. [Online]. Available: https://metacademy.org/graphs/concepts/weight_decay_neural_networks. [Accessed: 12- Apr- 2017].

[39] Traffic-Sign Detection and Classification in the Wild Zhe Zhu, Dun Liang, Songhai Zhang, Xiaolei Huang, Baoli Li, Shimin Hu, IEEE Conference on Computer Vision and Pattern Recognition (CVPR),2016

[40] U, Zakir, Edirisinghe E, and Leonce A. "ROAD SIGN SEGMENTATION BASED ON COLOUR SPACES: A COMPAR TIVE STUDY". International Conference, Computer Graphics and Imaging 11.1 (2010): 72 - 79. Print.

[41] "Image Structure", Www-visl.technion.ac.il, 2017. [Online]. Available: http://www-visl.technion.ac.il/~re/anatlab/html/4-Color/sec1_4.html. [Accessed: 27- Jun- 2017].

[42] "Color Models | Intel® Software". Software.intel.com. N.p., 2017. Web. 13 June 2017.

[43] García-Garrido, M. Á., Sotelo, M. Á., Martín-Gorostiza, “Fast road sign detection using Hough transform for assisted driving of road vehicles” In Proceedings of Computer Aided Systems Theory, pp. 543-548,2005

[44] "Feature Detectors - Sobel Edge Detector", Homepages.inf.ed.ac.uk, 2017. [Online]. Available: https://homepages.inf.ed.ac.uk/rbf/HIPR2/sobel.htm. [Accessed: 27- Jun- 2017].

[45] "Edge Detection With The Sobel Operator In Ruby". saush. N.p., 2017. Web. 19 June 2017.

[46] Shapiro, Linda and Stockman, George. "Computer Vision", Prentice-Hall, Inc. 2001

[47] C. F. Paulo and P. L. Correia, "Automatic Detection and Classification of Traffic Signs," Image Analysis for Multimedia Interactive Services, 2007. WIAMIS '07. Eighth International Workshop on, Santorini, 2007, pp. 11-11. doi: 10.1109/WIAMIS.2007.24

[48] "Traffic Sign Recognition", Vision.ee.ethz.ch, 2017. [Online]. Available: http://www.vision.ee.ethz.ch/~timofter/traffic_signs/. [Accessed: 03- Jul- 2017].

[49] Gottlieb, Allan; Almasi, George S. (1989). Highly parallel computing. Redwood City, Calif.: Benjamin/Cummings. ISBN 0-8053-0177-1.

[50] Klöckner, Andreas; Pinto, Nicolas; Lee, Yunsup; Catanzaro, Bryan; Ivanov, Paul; Fasih, Ahmed (2012). "PyCUDA and PyOpenCL: A scripting-based approach to GPU run-time code generation". Parallel Computing. 38 (3): 157–174. arXiv:0911.3456 Freely accessible. doi:10.1016/j.parco.2011.09.001.

[51] Gaster, Benedict; Howes, Lee; Kaeli, David R.; Mistry, Perhaad; Schaa, Dana (2012). Heterogeneous Computing with OpenCL: Revised OpenCL 1.2 Edition. Morgan Kaufmann.

[52] "Conformant Companies". Khronos Group. Retrieved April 8, 2015.

[53] Gianelli, Silvia E. (January 14, 2015). "Xilinx SDAccel Development Environment for OpenCL, C, and C++, Achieves Khronos Conformance". PR Newswire. Xilinx. Retrieved April 27, 2015.

[54] Howes, Lee (November 11, 2015). "The OpenCL Specification Version: 2.1 Document Revision: 23" (PDF). Khronos OpenCL Working Group. Retrieved November 16, 2015.

[55] Gaster, Benedict; Howes, Lee; Kaeli, David R.; Mistry, Perhaad; Schaa, Dana (2012). Heterogeneous Computing with OpenCL: Revised OpenCL 1.2 Edition. Morgan Kaufmann.

[56] Radford Juang OpenCLv0.17, "OpenCL Toolbox v0.17 - File Exchange - MATLAB Central", Uk.mathworks.com, 2017. [Online]. Available: https://uk.mathworks.com/matlabcentral/fileexchange/30109-opencl-toolbox-v0-17. [Accessed: 25- Aug- 2017].

[57] H. Kaeslin, Digital Integrated Circuit Design: From VLSI Architectures to CMOS Fabrication, 1st ed. Cambridge University Press, 2008 (cit. on pp. 13, 14).

[58] Xilinx Inc. (2016). Xilinx: What is an FPGA? Field Programmable Gate Array, [Online]. Available: http://www.xilinx.com/training/fpga/fpga- field- programmable- gate-array.htm (visited on Jul. 19, 2016) (cit. on p. 13).

[59] Opencl overview. http://www.khronos.org/assets/uploads/developers/library/overview/opencl-overview.pdf/.

[60] J. Tompson and K. Schlachter, An Introduction to the OpenCL Programming Model. New York: New York University, 2017, pp. 1-8.

[61] NVIDIA, 2012. Opencl programming guide for the cuda architecture, version 4.2. http://developer.download.nvidia.com/compute/DevZone/docs/html/OpenCL/doc/OpenCL_Programming_Guide.pdf.

[62] Stone, John E.; Gohara, David; Shi, Guochin (2010). "OpenCL: a parallel programming standard for heterogeneous computing systems". Computing in Science & Engineering. doi:10.1109/MCSE.2010.69.

[63] BOYDSTUN, K., 2011. Introduction opencl (caltech lecture). http://www.tapir.caltech.edu/˜kboyds/ OpenCL/opencl.pdf.