VNiVERSiDAD
D SALAMANCA

Efficient Object Detection in Mobile and Embedded Devices with Deep Neural Networks

Author:

William Vigilio Raveane

Supervisor:

María Angélica González Arrieta

Doctoral Thesis submitted for the degree of Doctor of Philosophy (Ph.D.) in Computer Science

October 20, 2020

Declaration of Authorship

María Angélica González Arrieta, full professor of the Computer Science and Artificial Intelligence area at the University of Salamanca,

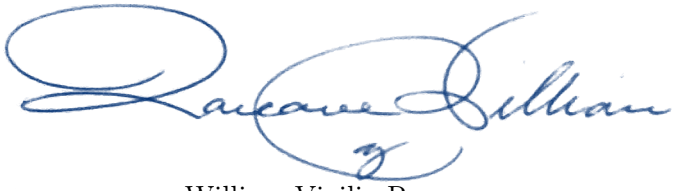
CERTIFIES

That the present document, entitled **Efficient Object Detection in Mobile and Embedded Devices with Deep Neural Networks** has been prepared under her supervision at the Computer Science and Automation Department of the University of Salamanca by **William Vigilio Raveane**, and constitutes his thesis for the degree of Doctor of Philosophy (Ph.D.) in Computer Science.

GONZALEZ ARRIETA MARIA ANGELICA - 06555917C

Firmado digitalmente por GONZALEZ ARRIETA MARIA ANGELICA - 06555917C
Nombre de reconocimiento (DN): c=ES, serialNumber=IDCES-06555917C, givenName=MARIA ANGELICA, sn=GONZALEZ ARRIETA, cn=GONZALEZ ARRIETA MARIA ANGELICA - 06555917C
Fecha: 2020.10.22 13:10:50 +02'00'

María Angélica González Arrieta



William Vigilio Raveane

Date: October 20, 2020

Acknowledgements

Foremost, I would like to express my deepest gratitude to my thesis supervisor Dr. Angélica González Arrieta for her continuous support during these years. She has helped me in more ways than I can count, and I am extremely grateful to have the opportunity to do my thesis with her.

Furthermore, I would like to extend my sincere gratitude to Dr. Heiko Neumann, for providing me the opportunity to conduct a research visit in the Institute of Neural Information Processing at Ulm University. It was an invaluable learning experience.

I also want to express my deepest appreciation to Dr. Tom Marvin. He introduced me to the amazing world of neural networks in the most unexpected of ways. His many teachings in Physics and his great wisdom has had a large impact on me and the way that I think.

Most important of all, I would like to give my biggest thanks to the person that has always been there for me, has always supported me, has always listened to me, and has always taught me so many things in life. She made me the person I am today, and I would not be here without her. For her dedication, her understanding, her patience, and her love, I dedicate this thesis to her, my Mother. Thank you!

Abstract

Neural networks have become the standard for high accuracy computer vision. These algorithms can be built with arbitrarily large architectures to handle an ever growing complexity in the data they process. State of the art neural network architectures are primarily concerned with increasing the recognition accuracy when performing inference on an image, which creates an insatiable demand for energy and compute power. These models are primarily targeted to run on dense compute units such as GPUs. In recent years, demand to allow these models to execute in limited capacity environments such as smartphones, however even the most compact variations of these state of the art networks constantly push the boundaries of the power envelop under which they run. With the emergence of the Internet of Things, it is becoming a priority to enable mobile systems to perform image recognition at the edge, but with small energy requirements.

This thesis focuses on the design and implementation of an object detection neural network that attempts to solve this problem, providing reasonable accuracy rates with extremely low compute power requirements. This is achieved by re-imagining the meta architecture of traditional object detection models and discovering a mechanism to classify and localize objects through a set of neural network based algorithms that are better aimed to mobile and embedded devices.

The main contributions of this thesis are: (i) provide a better image processing algorithm that is more suitable at preparing data for consumption by taking advantage of the characteristics of the ISP available in these devices; (ii) provide a neural network architecture that maintains acceptable accuracy targets with minimal computational requirements by making efficient use of basic neural algorithms; and (iii) provide a programming framework for how these systems can be most efficiently implemented in a manner that is optimized for the underlying hardware units available in these devices by taking into account memory and computation restrictions.

Contents

List of Figures

List of Tables

1 Introduction

Biological visual systems are easily able to recognize objects due to large amounts of specialization and adaptation through natural evolution. These systems are interconnected with other parts of the nervous system, such as memory and reasoning centers in the brain, so the task of vision is hardly something that can be separated and resolved in a standalone manner. However, this is what computer vision systems today try to achieve, primarily because the subtle complexities of biological vision is extremely difficult to replicate. However, it is still possible to take inspiration on how certain aspects of naturally evolved vision works, to mimic simplistic functionality.

Initially, computer vision systems took a very synthetic approach to the problem. Image recognition algorithms such as SIFT [1], SURF [2], or HOG [3] are based on manually tuned feature descriptors, and although they are loosely based on biological principles, such as sensitivity to certain orientations, they continue to be based mostly on geometric qualities in the image and easily fail as the complexity of shapes and forms varies. In recent years, however, it has become evident that a better approach is necessary. One of the most successful of these has been biologically inspired deep networks [4], where input images are processed through several layers of feature extractors and are eventually classified as one out of several target labels. These works are based on the findings by Hubel and Wiesel [5] where two main types of cells were identified in the primary visual cortex of cats, the so called simple cells and complex cells, connected through a cascading hierarchical model. The first functional model to be inspired by these findings was the Neocognitron, introduced by Fukushima [6], wherein two types of cells similarly take on the roles of local feature extractors to achieve tolerance against deformation.

Later work by LeCun [7] introduced the convolutional neural network (CNN) as a more robust system for the identification of handwritten characters in scanned documents. Again, a similar dual cell type system is used, where a convolutional layer extracts features and a subsampling layer reduces the input space to tolerate the space-wise variations of these features. These convolutional neural networks

form the foundation for most modern image recognition systems today, which are based on hierarchical deep learning techniques, and they have proven to be a powerful solution when applied to a large variety of complex recognition tasks.

The power of convolutional neural networks lies in the ability of the network to learn an optimal set of filters able to extract the most distinguishing features present in an image training data set. As with regular neural networks, when this process is performed over various consecutive layers, a strong non-linearity in the sample data can be modeled by these complex networks. The network is then capable of successfully classifying images it had never been exposed to previously during its training phase. The capacity to generalize the model in this manner allows them to have large flexibility in the characteristics of the image data they can successfully classify.

As a result, systems based on convolutional neural networks started to continuously win multiple academic image recognition competitions, such as large scale image recognition [8], handwritten digits [9], and traffic signs [10], wherein they always outperform considerably the results obtained by other types of systems – and in some cases even the results obtained by a panel of humans that have been subjected to an identical classification task over the same test data. Today, there is little doubt that CNNs have become the standard for perception based computer vision.

The problem, however, is that these systems are computationally expensive, and must be deployed on large compute nodes. It is therefore very challenging to fit the algorithms required for this kind of task into a mobile environment where computing capacity is limited. The main focus of this work is to demonstrate how a CNN based system can be implemented in low energy environments like mobile and embedded devices, while still being capable of performing the complex task of real time object detection.

1.1 Contributions

The specific contributions introduced in this work are as follows:

1. An algorithm to pre-process images to extract salient features while maintaining a normalized data distribution which is adapted to the image pipeline of camera based devices.

2. A neural network architecture framework to design models that require minimal compute resources to perform recognition in low energy environments.
3. An algorithm to share the activation maps of a classifier backbone in order to extend the classification task into an object detection task.
4. An algorithm to combine the results of spatially distributed inference results to find discrete objects.
5. An ensemble-based inference methodology to provide context information and increase the robustness of detections.
6. A description of how to most efficiently implement the proposed neural network by better targeting the available hardware resources in mobile and embedded devices.

1.2 Organization

In this work, an in-depth look is given at the development of a CNN based computer vision system, with special emphasis on its implementation in low power mobile and embedded portable devices.

In Chapter 2, a brief review of neural networks, and its application to computer vision through convolutional neural networks. The types of layers commonly used for this kind of network are described, and the connection between CNNs and biologically inspired computer vision systems is established.

The primary topic of this thesis appears in Chapter 3, where the proposed neural network architecture is described, along with the corresponding algorithms to tackle the task of object detection in low power environments. The chapter begins explaining the imaging pipeline of mobile devices and describes how to adapt the image pre-processing algorithm of contrast normalization to this environment. Next, the basic architecture of the backbone classification network is provided, with an explanation on certain design elements that make it more applicable to mobile devices. This chapter then explains how this backbone can be extended to handle to problem of localization by adapting the network architecture using shared convolutional maps. Finally, an algorithm for analyzing and collapsing multiple coalescing detection results on a shared map is provided, which is a more mobile friendly alternative to what is utilized in state of the art object detection networks.

Next, Chapter 4 gives an overview of the hardware and environment characteristics of mobile and embedded devices. Information on the compute units found in such hardware is reviewed, along with a discussion on how memory access routes can affect the implementation of these systems. This is followed by a detailed explanation of different optimization methods that can be used to implement the software that runs the proposed object detection network.

Chapter 5 provides the experimentation methodology and gives the results obtained. A sample baseline application and its corresponding neural network model are described. This application is then used as baseline to perform several experiments on, which help to establish the correctness of the proposed methods. Chapter 6 follows a similar objective but on a more detailed real world application use case. This chapter follows the development of an ear recognition application from start to finish, explaining how the proposed method allows the implementation of such a system in a compact portable hardware device.

The work finishes with Chapter 7 providing the conclusions of this work and raises some points of discussion and future lines of research.

2 Background

Computer vision describes the activity where a machine observes the real world through an image capture device and extracts useful and actionable information from it. It is therefore an important interface between the world and the machine, becoming a vital stepping stone for agents to display artificial intelligence capabilities.

Before treating the subject in detail, it is important to understand more about computer vision and its relationship to artificial intelligence.

2.1 A History of Deep Learning

The field of computer vision spans from the simple extraction of features to infer basic pieces of information, up to fully understanding and decoding of the visual content of a scene. This understanding is the natural first step towards reasoning and deduction, which marks one of the characteristics of a general artificial intelligence acting in the real world.

Defining intelligence itself is a tricky matter, multiple definitions have been proposed over the years, some examples that stand out might be:

> By "general intelligent action" we wish to indicate the same scope of intelligence as we see in human action: that in any real situation behavior appropriate to the ends of the system and adaptive to the demands of the environment can occur, within some limits of speed and complexity. [11]
>
> Intelligence is the capacity of a system to adapt to its environment while operating with insufficient knowledge and resources. [12]

These two definitions allude to two very interesting recurring characteristics: (i) intelligence mimics human behavior, and (ii) intelligence is action that is bounded by limited resources.

Given that the only fully working model of intelligence that is known is human behavior, it is natural that this becomes the ideal upon which artificial agents can model their own functions. The human brain is therefore the epitome of intelligence, and understanding its functionality can ultimately lead to discovering principles that can be either directly or indirectly applied to its synthetic counterparts.

The brain, in essence, is a large network of neurons. This simplified view of the organ has formed the basis for one of the potential solutions to artificial intelligence: Artificial Neural Networks (ANN). In fact, many of the most important advancements in artificial intelligence stem from basic neural networks which simulate (albeit at a high level) the complexity of biological neurons. These are mathematical models that exhibit similar patterns in functionality as their natural analogues. At the same time, observations from the implementation of ANNs have made their way back into neuroscience to help better understand and model the human brain. As a result of this cross-fertilization, the field of computational neuroscience has become an important interface between these two very distinct disciplines, with much important research originating from this area seeing its application into novel concepts in machine vision.

A deep neural network (DNN) is a specialization of the ANN, as it advances the modeling of the brain by adapting a hierarchical extraction and flow of information. This hierarchy of information and processing can be most notably seen in the visual cortex of the brain. Some of the pioneering research into neuroscience carried out over half a century ago was precisely to better understand this hierarchical structure in the visual cortex of mammals. Inspiration from these findings found its way into the development of an even further specialization of DNNs, the convolutional neural network (CNN). The CNN is a type of network that is specifically built for visual tasks by virtue of the mathematical modeling of the functionality seen in the so-called “simple” and “complex” cells of the visual cortex.

Although ANNs first appeared as a research direction many decades ago, the field as a whole came to a stagnating halt after it became apparent that regardless of their basis on biology, the these networks had many hard to solve difficulties modeling the real world. As a result, research in ANNs stopped for the most part at the beginning of the century, with almost no new applications making use of them for well over a decade. During this time, computer vision tasks returned to the tried and true practice of engineering hand-made features and inferring from them via shallow machine learning classifiers.

However, in 2012, a sudden explosion of interest in ANNs was sparked by the results of that year's ImageNet challenge [8], the first time a CNN-based system outperformed, by a large margin, the results from traditional computer vision methods. This single event is seen by many as the spark that initiated the current artificial revolution we are currently undergoing, as it resulted in an outpour of research and applications of DNN-based systems into computer vision tasks, but also to many other areas of Artificial Intelligence. This newfound interest in neural networks has initiated an entire new field which has come to be known as Deep Learning.

2.2 The complexity of computer vision

The Convolutional Neural Network (CNN) [13] has become a general solution for image recognition with variable input data. CNNs consist of two stages – one for automated feature learning, and another for classification – both of which can be successfully trained in tandem through gradient descent of the error surface [14]. Its results have consistently outclassed other machine learning approaches in large scale image recognition tasks [15], outperforming even human inspection of extensive datasets [16].

Compared to other feature-based computer vision methods such as SIFT [1] or HOG [3], CNNs are much more robust and tolerant to shape and visual variations of the images or objects intended to be recognized. However, contrary to such methods, an execution of a CNN will only recognize features on a single image block of size equal to the input dimensions of the network. As CNNs are usually trained with small image patches, this recognition area is likewise small. As a result, to run image recognition over a larger image size, it is necessary to repeatedly apply the same network over multiple regions. This is a very common technique named sliding windows, albeit a time consuming one as the execution time naturally grows in proportion to the number of sampled blocks.

With the increasing use of embedded hardware, it has naturally become a priority to endow mobile devices with computer vision capabilities. The use of CNNs, when applied through a sliding window methodology, allows a large range of important image recognition tasks to be carried out, many of which would have a great impact on the everyday usage of mobile hardware by end users. Some examples of this are text recognition [17] for visual language translators, human action [18] and face

[19] recognition for greater user interactivity with social applications, or even traffic sign recognition [20] for embedded automotive applications. The unique task of logo recognition is taken as a sample usage of mobile implemented CNNs in this work, something which would have large opportunities for commercial applications to increase company brand loyalty, perception and awareness among consumers, depending on the context it is used in. However, the same methods and network architecture described here would be equally applicable to solving similar problems, such as those described above.

Due to the high computational requirements of a CNN, the need for mobile computer vision has traditionally been met by outsourcing image analysis to a remote server in communication with the device over an internet connection. This approach, while effective, introduces large delays and is hardly an appropriate solution when user interactivity and real-time responsiveness are paramount. As embedded hardware capacity continues to grow with each new generation of low energy processors, this trend has gradually shifted towards implementing image recognition algorithms on the device itself with all computations carried out locally. Regardless, these devices continue to display performance limitations, as well as having intrinsic architecture constraints which result in slow memory access. It is therefore important to find new possible optimizations, so as to better utilize the computational power of the device.

2.3 Deep Convolutional Neural Networks

This chapter provides the background material for neural networks, making an emphasis on the biological concepts behind these mathematical models, and how they can be applied to supervised learning tasks in machine learning.

2.3.1 Neural Networks

Artificial neural networks consist of a machine learning model inspired by the structure of biological neurons in the brain, and the connective networks they form. They are constructed through a connective network of independently operating cells, through which information flows from one end of the network to the other, being processed in a different manner at each node it passes through in accordance to the activation behavior of that neuron. They are modelled after natural neurons

in the human brain, and although the behavior of brain cells is far more complex, the basic idea behind the biological version can be transferred quite sucessfully to a mathematical representation.

Biological Fundamentals of Neural Networks

Neurons in the brain are connected to each other through synapses, connections usually formed from the axon of an emitting cell to the dendrites in the receiving cell. The synapses transmit electrical and chemical signals from one neuron to another through a process called synaptic transmission [21]. Every neuron connection forms a presynaptic – postsynaptic relationship with each other, where the former is the neuron transmitting a signal, and the latter is the neuron receiving the information. Note that a neuron that is postsynaptic can itself also be presynaptic to yet another cell, thereby building networks of chained connections formed by the sequential linking of cells.

When a neuron does not receive any external stimulation, it holds a negative rest potential voltage which has been adapted to an optimal value for the specific task the cell carries out. Different neurons will usually have very different potentials, though usually negative. A neuron is stimulated when it receives a synaptic transmission from another cell connected to its dendrites. If the incoming transmission is strong enough to overcome the rest potential and reach a predetermined threshold, the neuron will trigger one or more spikes that are then transmitted to all other neurons connected to its axon. The voltage level of the spike is constant, but the frequency and number of spikes transmitted will be regulated by the neuron in a manner proportional to the combined stimulation it received.

The connections formed by a synapse between two cells can have either an excitatory or inhibitory effect on the receiving neuron's stimulation. An excitatory synapse will act as previously described increasing the neuron's potential voltage. An inhibitory synapse, on the other hand, will act in the opposite manner by further lowering the neuron's potential – thereby making it more difficult for other incoming transmissions to overcome it and reach the required threshold value needed to trigger a spike. As there may be both types of synapses present on the receiving end of each cell, complex patterns of excitatory and inhibitory stimulations can result in different behaviors altering the signals this cell itself transmits to others.

The system portrayed here describes the basic system by which cells communicate

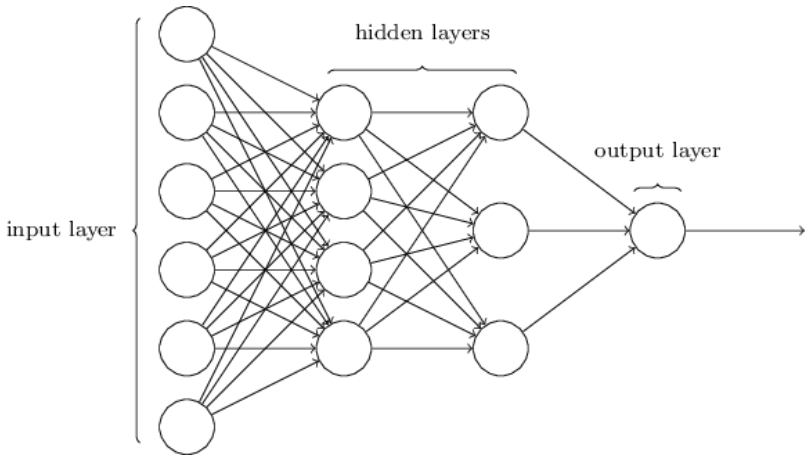


Figure 2.1: Basic structure of a feed-forward MLP neural network.

with one another, as well as the manner in which information can be either forwarded or restrained throughout the connected network – depending on the unique behavior governing the actions of each individual neuron. Although this is a very simplified representation of what in reality is a much more complex nervous system, it is possible to create an artificial model that mimics these basic properties, thus forming the core of what an artificial neural network attempts to achieve.

Mathematical Model of a Neural Network

An artificial neural network is modeled as multiple neurons arranged in consecutive layers, where the neurons in one layer are only connected forward to other neurons in the immediately adjacent layer. This forms a feed-forward connective architecture, where data flows in only one direction throughout the network. This simplistic feed-forward version of the system portrayed is also called a Multi Layer Perceptron (MLP), and its basic structure is shown in figure 2.1.

An individual neuron in the network is modeled as a linear combination of its own bias level plus the individually weighted input signals coming from neurons in the previous layer. The bias is a constant amount, set during training, and unique to each neuron. In a way, this mimics the potential voltage of natural cells, acting as a resistance against the stimulus received from other neurons. In the mathematical model, however, the bias is not limited to negative values which can lead to the

bias acting as support to other neuron's stimulations.

Stimuli from other neurons is combined linearly, proportional to weights that have been pre-established during training as optimal for an efficient combination of signals. The weights for each connection can take on any value, either positive or negative, causing the connection to have an analogous excitatory or inhibitory synaptic effect to the receiving neuron's stimulation.

The output of a neuron is thus given by:

$$y_j = g\left(b_j + \sum_i w_{ij} y_i\right) \tag{2.1}$$

Where i represents each of the neurons in the previous layer that connect to the currently computed neuron j, such that y_i is the output value of each neuron in the preceding layer, modulated by the values w_{ij} corresponding to the weight of each corresponding connection. The bias of the current neuron is given by b_j. The function $g(x)$ is a nonlinear activation function, resulting in the neuron's final output which is assigned to y_j.

Before transmitting this stimulus combination to other neurons, the calculated value is passed through a non-linear activation function. Similar to the natural system, the activation function acts as a filter to regulate the output of the neuron. The mathematical model uses a continuous function such as a hyperbolic tangent to achieve this, the primary reason being that this function is easy to derive, a property that greatly aids the training process. Additionally, the interval of values produced by the hyperbolic tangent function is in $(-1, 1)$, which helps to avoid the output values from blowing out of proportion if too strong stimuli were received as input.

$$g(x) = tanh(x) = \frac{e^{2x} - 1}{e^{2x} + 1} \tag{2.2}$$

$$\frac{dg}{dx} = 1 - g(x)^2 \tag{2.3}$$

The combined effect of multiple neurons all acting in this manner, each one contributing varying output values modulated by a weight in each connection, all trying to overcome or reinforce a unique bias value for each receiving neuron, and

then re-transmitting a signal unto the next layer of neurons – is the basis by which MLPs are formed and how information travels through the network.

If an MLP is structured correctly, and its parameters are tuned to the right configuration, the network can work as a computing machine capable of receiving input data, processing it internally, and producing a result. This is achieved by keeping open neurons on the first (input) layer where data can be written directly and transmitted to the following layer. The next (hidden) layer can process and manipulate the data, and re-transmit it to either additional hidden layers or to the network's final stage. The last layer (output) layer consists of neurons whose individual activated outputs correspond to the final output of the network, thus providing the results of the information processing. Although this is a different computing paradigm than that of a traditional Von Neumann architecture, it is quite valid and capable of executing complex programs, such as the well known example of implementing the XOR operation, which is a simple but non-linear function, through an artificial neural networks consisting of a single hidden layer.

In particular, programs to model data are of specific interest and one of the tasks MLPs are better suited for – that is, to predict the ideal output values given a set of input data. The combination of multiple linear calculations over multiple layers is capable of modeling very complex non-linear data approximations, making MLPs one of the most powerful machine learning systems available.

The challenge arises in programming the MLP so that it behaves as desired. In addition to the inherent structure with which it is initially laid out, an MLP is parameterized by the weights in each connection and the bias values for each neuron. Adjusting these values will affect the behavior of the system, and if the correct combination of values is found, the model will be able to produce accurate predictions. This is achieved by training the neural network, a process involving the exposure of the network to multiple examples of input data along with the ideal outputs each data sample is expected to produce. One step at a time and for each data sample, the training procedure will adjust the network's parameters in an attempt to more closely match the desired target values. This is performed through the Backpropagation algorithm [22].

2.3.2 Neural Networks in Computer Vision

Artificial neural networks have also broad applications in computer vision. Again, following inspiration from biological systems – in this case on the functions in the

visual cortex of the brain – a mathematical model can be built that mimics the behavior of natural systems.

Hubel & Wiesel Visual Cortex Model

The work by [5] played an important role in the understanding of how the visual cortex operates, particularly the cells responsible for orientation selectivity and edge detection of visual stimuli in the V1 primary visual cortex. Two main types of cells were identified here, having elongated receptive fields and therefore having a better response to elongated stimuli – such as lines and edges. These cells received the names of *simple* and *complex* cells.

Simple cells have distinct excitatory and inhibitory regions, both forming elementary elongated patterns in one particular direction, position and size for each cell. If visual stimulation reaches the cell in the exact same orientation and position such that it aligns itself perfectly with the patterns created by the excitatory regions, and at the same time the inhibitory regions do not get stimulated – the neuron gets activated and responds accordingly.

Complex cells operate in a similar manner. Like simple cells, they have a particular orientation they are sensitive to. However, they do not have position sensitivity. Therefore, a visual stimulation needs only to arrive in the correct direction to activate this neuron, regardless of the exact position it falls on.

Another important fact about the cells in the visual cortex is the structure that they form. Along the cortex hierarchy, proceeding from the V1 region of the visual cortex and proceeding to the V2, V4, and IT regions, one finds that the complexity of the ideal stimuli for each cell increases in complexity. Simultaneously, the activations of neurons become less sensitive to position and size. This happens as cells activate and propagate their own stimuli to other cells they are connected to in this hierarchy, primarily due to alternating simple and complex cells.

Mathematical Model of Convolutional Neural Networks

Most of the groundwork of the deep convolutional neural network (DCNN) was laid out by [7]. Originally intended for handwritten character recognition, these systems have evolved into more complex algorithms that are able to classify images from any source, as well as other similarly structured data. For DCNNs to be effective,

the input data should have strong local correlation, meaning that data points close in space should be related to one another by a causal relationship. Data should also be statistically invariant over space, otherwise explained as there being relatively similar values, ranges and deviations across the input so that features learned in one portion of the image are equally applicable elsewhere [23].

In fact, there is no requirement for the data to be two dimensional at all, evidenced by the many successful applications of this type of network with other data types. For exmaple, with 1D data as used in time series prediction [24], non-visual 2D data networks for spoken word spectra analysis [25], and even 3D data for video and motion classification [26] or volumetric recognition [27]. The rest of this work will center on the applications of DCNNs to visual stimuli and will therefore focus 2D data sets.

Traditionally, the architecture of image recognizers has always been divided into two separate functions. The first of these being a fixed feature extractor that transforms input images into a low dimensional representation vector holding characteristics found in the input image. The resulting vector should not only be easily comparable, but also relatively invariant to transformations and distortions in the input image image dataset. Next, a classification module takes this low dimensional vector and interprets it by classifying the data into one of several possible target labels.

The difficulty with such an architecture is the feature extraction stage, which usually requires manual fine-tuning of the parameters to find the features needed which can then be successfully classified. The advantage of DCNNs over previous systems, then, is the unification of both procedures into a single neural network system. Using backpropagation, the DCNN is able to automatically learn the parameters that will produce the most optimal set of feature extractors, as well as learn the optimal way to classify those same features into discrete target classes.

A DCNN is composed of several specialized layers, the first few of which form the feature extraction stage, and the latter compose a classifier. The basic architecure can be seen in figure 2.2. The feature extraction stage is built with alternating convolutional and subsampling layers.

This alternating structure between neuron types is influenced by the Hubel & Wiesel model of the visual cortex previously described in section 2.3.2. Convolutional neurons have similar properties to the biological simple cells, in that they are highly sensitive to the edges of an image, in particular to the orientation of

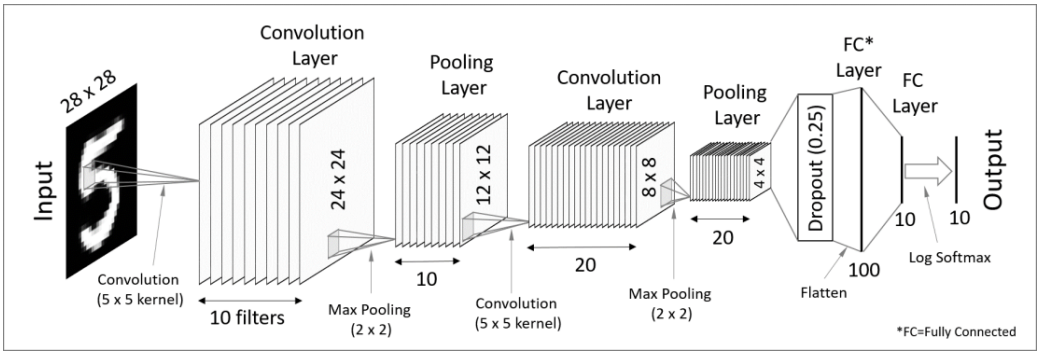


Figure 2.2: The usual architecture forming a deep convolutional neural network.

edges and the position where they appear in the image. Subsampling layers, on the other hand, are modeled after complex cells, in that the sampling procedure effectively *blurs* away position sensitivity by bundling together the activation values from neighboring pixels. Finally, the structure in the biological model is mimicked by the convolutional neural network as well, in that further layers increase the complexity of the neuron responses while increasing the tolerance for positional invariance.

Convolutional Layer

In the feature extraction stage, the single value neurons forming traditional MLPs are replaced by matrix processors which perform an action over the 2D image data passing through them, instead of a single scalar value. The output of each convolutional neurons is given by:

$$Y_j = g\left(b_j + \sum_i K_{ij} \otimes Y_i\right) \tag{2.4}$$

Compare with equation 2.1, noting that the output of the neuron Y_j is now a matrix, as are the outputs Y_i of each connected neuron in the preceeding layer, all of which are convolved ($\otimes$ operator[1]) with the corresponding kernel filters K_{ij} of each connection. These kernels replace the scalar weights of the MLP version

[1]Existing literature often makes use of the $*$ symbol to denote the convolution operator, instead of the $\otimes$ symbol. However, both of these refer to the same matrix operation.

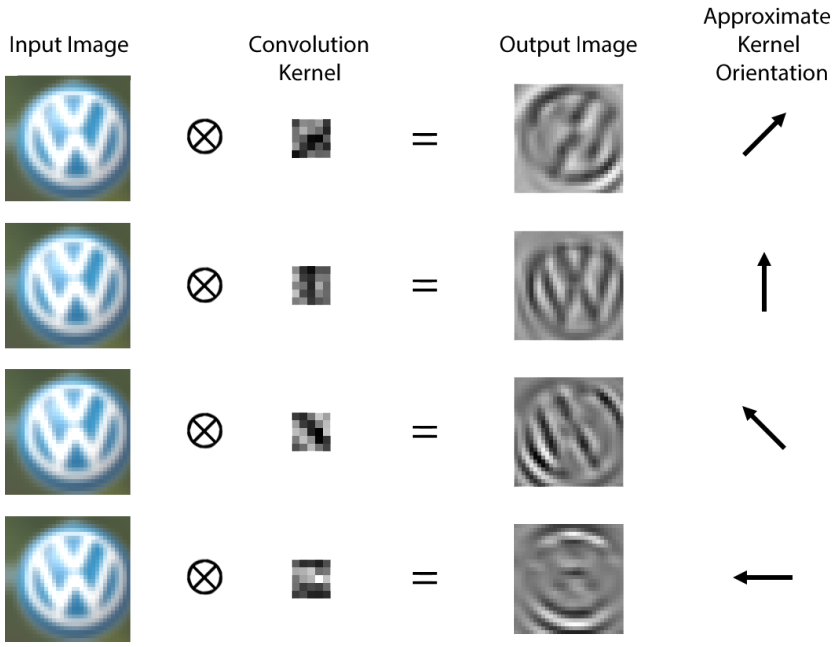


Figure 2.3: Example of performing convolution an image with different kernel filters, where it can be seen that each kernel enhances the edges that most closely resemble it's approximate orientation.

of the network. The term for bias for the receiving neuron b_j, and the nonlinear activation function $g(x)$ are identical to the original MLP equation.

The convolution operator has the effect of filtering the input image data with a previously trained kernel. This transforms the data in such as way that certain features, determined by the shape of the kernel, become more dominant by having a higher numerical value assigned to the pixels representing them. Kernels have specific image processing capabilities, and in fact, symmetrical and normalized kernels can convolve images for edge detection and other image processing tasks [28]. As can be seen in figure 2.3, however, the kernels learned by a DCNN will usually have more intricate shapes and structures to extract composite features which may have non-trivial purposes within the feature extraction stage of the neural network system.

The convolution operation is calculated iteratively over each pixel in the output data space. For each pixel, the matrix inner product is taken of the kernel with

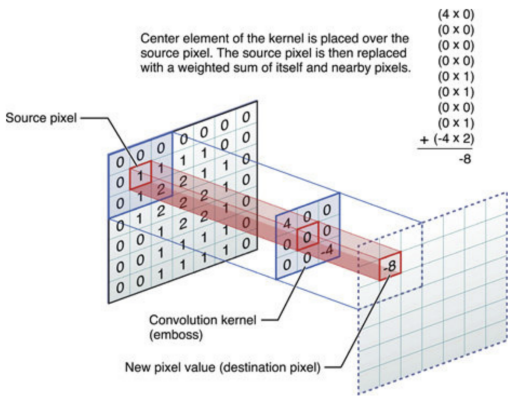


Figure 2.4: The convolution operation works by masking the input image with the same shape of the kernel at the corresponding position, a pointwise multiplication is then made and the results are summed together.

the corresponding masked area in the input image. A diagram of the operation is depicted in figure 2.4.

Other than this mathematical modification to handle bundled image data at each neuron, the operation of a neuron in a DCNN is essentially the same as that of a neuron in an MLP.

Max-pooling

In principle, DCNNs should have a small amount of tolerance over tiny perturbations in the input image data. For example, if two essentially identical images differ only by a translation over a few pixels, the network should be able to treat both identically, and not expect to learn an entirely new set of feature extractors to account for such a small divergence. This is achieved by reducing the sampling of the output from the convolutional neuron. By having a lower resolution, the same features which would otherwise differ only by a few pixels in their position, will now be assigned to the same subsampled output position.

The original specifications of DCNN suggested using simple subsampling (averaging) of the values in subsequent regions of pixels. However, it was demonstrated by [29], based on biological evidence on the functions of the visual cortex [30], that max-pooling is a much more effective method to reduce data size. Max-pooling effectively finds the maximum value over every consecutively sampled pixel window.

Figure 2.5: A 2:1 max-pooling operation applied on a 4×4 image reduces it to 2×2, with each new pixel corresponding to the maximum value of the subsampled pixels in the original image.

As a result, the data size of the input is reduced by a factor equal to the size of the window over which this function operates. Figure 2.5 depicts this operation.

Linear Classifier

The last stage in a DCNN is that of a linear classifier, which is exactly identical to the latter stages of a typical MLP.

The input of this classifier is given by the output of the last step in the DCNN feature extractor. By the time the data reaches this stage, the input images should have been distilled into a lower size representation of multiple extracted features, having been reduced several times over the course of several convolutional and max-pooling layers. This last stage should now be able to do a final non-linear classification over the feature vector to finally recognize the original input image. Therefore, the output layer of the classifier consists of one neuron for each of the target labels over which the data is being classified.

2.3.3 Computational complexity of CNNs

Convolutional Neural Networks are computationally expensive algorithms. This chapter analyzes the complexity of the forward pass of a CNN, thereby showing the importance of fast and efficient processing platforms for the computation of these systems.

The calculation of the activation maps Y_l for a layer l can be expressed as:

$$Y_l = g(W_l \otimes Y_{l-1}) \tag{2.5}$$

Where $g(\cdot)$ represents the activation function, and the $\otimes$ operation represents a layer-wide convolution operation. This process repeats for each of a total of L convolutional layers. Each of these layers has M_l neuron units, with weights W_l composed of convolutional kernels of dimensions $K \times K$, such that:

$$W_l \in \mathbb{R}^{K \times K \times M_l} \tag{2.6}$$

The activation maps for a particular layer, can be represented as:

$$Y_l \in \mathbb{R}^{H \times H \times M_l} \tag{2.7}$$

Where H represents the spatial dimension of the maps. The input data to the network is treated as the zeroth layer's activation map, that is, a 2D color image of square dimensions $H \times H$, and three color channels such that $M_0 = 3$. Assuming the use of padded convolutional operations and ignoring the effects of pooling or strided convolutions, the activation maps of all subsequent layers are assumed to have the same spatial dimensions as the input data for simplicity.

Using equation 2.5, it is possible to expand the computation for each activation map in a layer as:

$$Y_{l,n,x,y} = g(\sum_{m}^{M_{l-1}} \sum_{k_1}^{K} \sum_{k_2}^{K} W_{l,k_1,k_2}) \cdot Y_{x,y} \quad \forall x \leq H, y \leq H, n \leq M \tag{2.8}$$

This process is repeated for all layers, and it is therefore possible to see that the total number of multiply-and-accumulate (MAC) for a forward pass of a single layer can be expressed as:

$$N_{macs} = L \cdot M^2 \cdot H^2 \cdot K^2 \tag{2.9}$$

2.4 Mobile and Embedded Devices

Convolutional neural networks are computationally heavy and complex algorithms to execute. Normally, it is necessary to have a large compute budget in order to properly run such a network and obtain its results within a reasonable amount of time. This is even more important when performing inference over a sequence of images from a continuous video stream, as the results need to be ready within a similar amount of time as is available between one video frame and the next, otherwise the system will stutter and the frame rate will lag.

2.4.1 Cloud Computing vs Edge Computing

As a result of all this, state of the art computer vision systems usually run on large computer systems that consume a considerable amount of energy. This is usually the realm of computer vision systems running *on the cloud*, which is to say, in a large data center where energy and compute resources are available in abundance. However, these systems are far away from the source of the data, where the "distance" here is not just a metric of space, but also of the time and energy required to transfer the data from its source to the compute device. As a result, while the computation might be in abundance in such a system, the bandwidth required to maintain an active video data stream can put considerable tolls on the communication infrastructure to maintain such an application.

On the other hand, there is a large movement towards running such systems *on the edge*. This is a term utilized to explain that a particular system executes on one or more small devices deployed much closer to the source of the data. This offsets the pressure on the communication infrastructure and exchanges it towards requiring multiple compute resources to be available to handle the computations at the source. The communication between these edge devices and a central control system is now reduced to only a transfer of already processed results, rather than the full raw data. The major benefit of such a system is that it can be much more portable, as it doesn't rely on an expensive or fixed communication pipeline, and can instead be deployed virtually anywhere.

The difficulty that arises with edge computing, obviously, comes from requiring efficient computing resources, as small and inexpensive devices rarely have large compute capabilities, as they do not have access to large sources of energy, and they must often rely only on battery power or other low energy sources. The successful

deployment of an edge computing system, therefore, depends on the development of efficient algorithms that can make the best possible use of the limited available resources on such devices.

2.4.2 Mobile Devices

When describing edge computing, the most natural platform that can be described is the mobile phone. It estimated that over 5 billion people in the world have access to a personal mobile phone device, with over half of these devices being smartphones specifically. A smartphone is a very common platform that has the three essential elements required for such computer vision edge application: (i) a camera to act as a visual sensor and receive real world imagery, (ii) a microprocessor to act as a computation system on which to execute the computer vision algorithm, and (iii) a communication line to the internet to access virtually any other device in the world to transmit the results.

To facilitate things even more, the large majority of smartphones in existence use one of two operating systems: Google Android or Apple iOS. Either of these two platforms have common APIs that aid in making software development compatible across as many devices as possible. Furthermore, these two systems make use of *stores* which provide for an easy way to distribute and update the developed application to already deployed devices. As such, smartphones are a natural choice to deploy edge applications, as the system can make use of a widely available infrastructure that is already in place, thereby greatly minimizing the cost and effort required to deploy such systems.

Although a good choice, mobile phones also have their drawbacks, and this comes primarily in the form of device fragmentation. In this context, fragmentation refers to an ever growing variety of different types of smartphones in the world. It is estimated that there are over 60 thousand different variations of smartphones in the world today. Although operating systems and software frameworks are very common among them, the hardware capabilities of these devices obviously vary enormously, and it becomes impossible to predict how all devices will react when running a particular application. The three requirements for edge computing, while usually always present, may be extremely different between one smartphone to the next, having an unprecedented effect on the performance and accuracy of a computer vision model that might execute on it.

2.4.3 Embedded Devices

An embedded device usually refers to a full computing system, including some kind of microprocessor, memory, and input/output peripherals that are all deployed within the same integrated circuit or circuit board. These systems are usually purpose built, and support only a very specific application subset. As opposed to smartphones, these devices may not always be suitable for general computing tasks, but on the other hand, they can usually be customized with characteristics that are specific to the application task. Their use cases can vary wildly on a large number of different electronic applications, ranging from music players, watches, traffic light controllers, PLCs, medical imaging systems, or even avionics. Under some definitions of the term, a smartphone could also be considered an embedded device itself. However for the purposes of this work, a distinction is maintained between both platform types.

In the context of edge applications, embedded devices usually refers to low energy and portable variations of such systems, usually with on-board sensors such as embedded cameras, and some kind of networked I/O. An ever finer subset of these devices might be those that are targeted for *Edge AI* applications. This is a type of device that will often have some type of compute block that accelerates MAC (multiply and accumulate) computations, which form the basis of the large majority of linear algebra algorithms, and by extension, machine learning and neural network computations.

Some commonly available examples of these Edge AI embedded devices are those based on the hardware platforms: (i) Intel Movidius VPU, (ii) NVIDIA Jetson GPU, and (iii) Google Coral Edge TPU. Each of these platforms define specific accelerators (VPU, GPU, and TPU) that are capable of efficiently executing parallel MAC computations, usually within the energy requirements ranging between 0.5 to 15 Watts. This can be considerably less than a standard smartphone, while providing a larger compute throughput for MAC operations due to their specialized accelerators. These three platforms are often found in publicly available development kits with accessible SDKs, facilitating the entry point for developing and deploying prototype applications based on such platforms.

Naturally, there exist many other such devices, which are more production oriented, or more specific to a particular task or industry. An example of this would be in the field of Advanced Driver Assistance Systems (ADAS) and Autonomous Driving (AD), where far more specialized and robust systems exist for those particular uses

cases. Some examples might be Renesas V3H, Texas Instrument TDA4, and Xilinx ZYNQ, which are based on even more specialized ASIC or FPGA microprocessors to accelerate the types of algorithms often found in automotive perception and AI systems that require the use of cameras, radars and other sensors.

This work will focus on the Intel Movidius VPU platform as an embedded device baseline. This system was chosen as it's one of the most power efficient platforms from those previously described, requiring only 1 watt to power the VPU microprocessor, while achieving a neural network inference throughput that exceeds by far what can be obtained with a standard smartphone. Furthermore, this is the platform of choice to provide smart computer vision features to the widely available DJI aerial drones, which can open up very interesting edge applications.

3 Efficient Object Detection Neural Network

The main contribution of this work is to provide an architecture framework upon which a highly efficient object detection neural network can be built, such that the inference compute requirements are minimal and can execute efficiently in limited compute environments. This chapter explores the architecture proposed and details the motivation for the choices made in the architecture design and algorithms utilized. Great attention is placed on the imaging pipeline, to understand how to better devise an image pre-processing algorithm that is better suited to these hardware environments. Based on the results of this algorithm, a classification CNN architecture is proposed as primary backbone to carry out feature extraction tasks. This network is then adapted to extend its inference spatially and share its activation maps with neighboring locations in the image such as to create a localization algorithm. The localization information from the network must be further processed to collapse it into individual object detection results, for which a discrete inference algorithm is adapted and described.

3.1 Motivation

While the fields of artificial intelligence, computer vision and neural networks continue to advance, requiring always bigger and more complicated models, there is always room for ultra low energy devices to do inference with miniature models at the lowest end of the spectrum. These devices can then be deployed in large scale at low cost for applications such as Internet of Things and Edge AI.

Usually such miniature models are limited to processing basic sensor information that can be expressed in the form of a time series or other variable scalar amounts. The challenge is in developing model architectures that are capable to perform inference on more complex types of data, such as the task of computer vision on

images. Computer vision models still require complicated and computationally expensive architectures.

The computer vision task resolved by this work is specifically that of object detection. This task processes a large input image and detects within the image space one or more objects of interest, creating as its inference result a set of bounding boxes, each of which is marked with an object type (or class), a classification confidence, and the coordinates that determine the position of this object within the image space.

Presently, object detection is considered a mostly solved task, by the availability of architectures such as RCNN [31] and its variations (Fast RCNN [32], Faster RCNN [33], and Mask RCNN [34]), which breaks down the problem into a two-stage solution: (i) the first stage acts as a region proposal network to "find" possible objects of interest, while (ii) the second stage fine tunes these regions and properly classifies and fine tunes their localization.

Similarly, single stage detectors have also been developed which solve the problem in a single pass. Examples are SSD [35] and YOLO [36] plus its variations (YOLO v2 [37], YOLO v3 [38]). In this case, a more intricate architecture is developed to help with the multiple scales objects can appear at in the image. The main part of the network acts as a feature extractor, and two heads in the end the network act on these features at different scales to perform per-anchor classification and localization regression.

The two-stage models usually have greater accuracy while the single-stage networks run faster. However, both of these architectures have a common flaw: They rely on a heavy feature extractor backbone to power the object detection feature. While this backbone can be swapped with smaller and more efficient networks, there is a limit of diminishing returns that doesn't allow these meta-architectures to perform properly if the backbone network is too small. As such, it is not practical to run these networks in ultra low energy devices.

These state of the art systems all compete with each other obtaining ever improving accuracy metrics. And while there is considerable effort being placed on their efficiency for low energy devices, most of these improvements come from reducing the size of the backbones [39] or creating better interconnections between the layers to improve the learning through better flow of gradient information [40]. This helps such networks have extremely high detection and recognition accuracy on a large variety of object types, with extremely high variability in the data.

Another drawback of these systems is that they have been built with generalization of general object datasets [41] in mind. That is, they are intended to perform well for detecting well regardless of the hardware configuration they are running on. Therefore, little attention is paid to the preparation of the input data, and the network itself must spend more compute resources processing the image and extracting low level feature information, something which could otherwise be aided with the underlying imaging hardware available to the system.

When deploying a computer vision model in the domain of edge computing, however, you can bypass some of these design requirements. Primarily, the types of objects an edge object detector might be required to detect could be extremely simple to power basic IoT applications. So a neural network is not required to have expensive network backbones to process everything from cats and dogs to different types of vehicles, as is the case of the benchmark datasets that most state of the art object detection networks are developed for. Furthermore, when deploying to a particular hardware device (or type of device), it is possible to design architectures that more efficiently make use of available hardware features, both for image processing as for the network inference algorithms themselves.

This work therefore explores the development of a specialized neural network architecture that can be deployed with minimal compute resources on low energy mobile and embedded devices. The novelty of this system is rooted on several concepts which are explored in detail in this chapter.

3.2 Imaging Pipeline

This section explores the traditional hardware imaging pipeline for camera-based devices. Understanding how this pipeline works allows making better decisions on the image processing algorithms used to prepare the input image for consumption by the neural network.

3.2.1 Camera Sensor

The main component of a camera device is the image sensor. The components of camera sensor are sensitive to a wide range of wavelengths, and as such, light of any color can activate the sensor, making it impossible for a sensor to differentiate among different colors in the light that reaches it. The simplest way to reconstruct

color information is to split the incoming light through a *Philips* prism and redirects it to three separate sensors, each one treated with a color filter to limit their sensitivity to a narrower wavelength bandwidth in each of red, green and blue wavelength ranges. The three images from these sensors can then be put together to form a traditional 3-channel RGB image. This method is depicted in Figure 3.1. While very effective and the result is a very high quality color image, this system requires multiple sensors and extremely precise optical paths, which results in an increased hardware cost. As such, this system is not very ideal for a mass deployment of camera-based hardware devices.

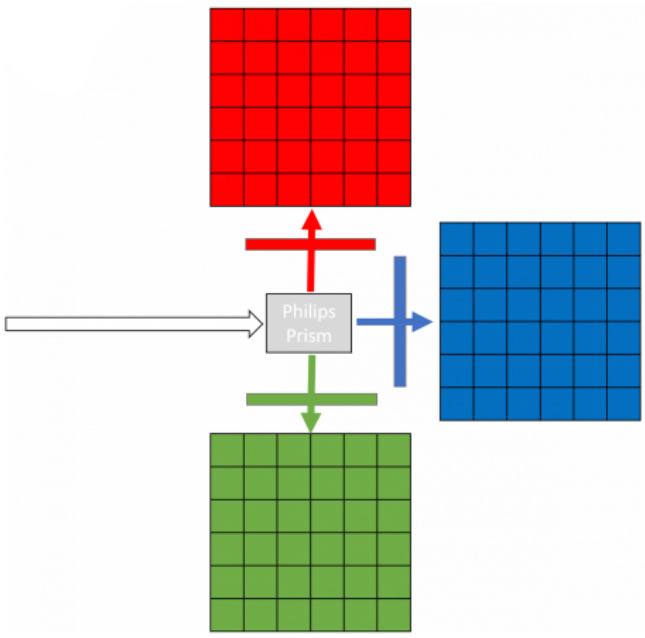


Figure 3.1: A 3-CCD sensor system used to maintain create quality color images with the help of a prism that separates the optical path in three for each of three sensors, each with a different color filter. The result is three individual images that then correspond to to each of the three color channels of a final image.

As a result, an alternative method exists which makes use of a single sensor to form a color image, thereby greatly reducing the manufacturing cost. This method consists of a single photosensor array that instead of being treated with a single color filter, it has a Color Filter Array (CFA) forming a Bayer Pattern. This filter

array is a grid pattern where each pixel has different color sensitivity, such that the resulting image interlaces pixels of different colors. This system is shown in Figure 3.2.

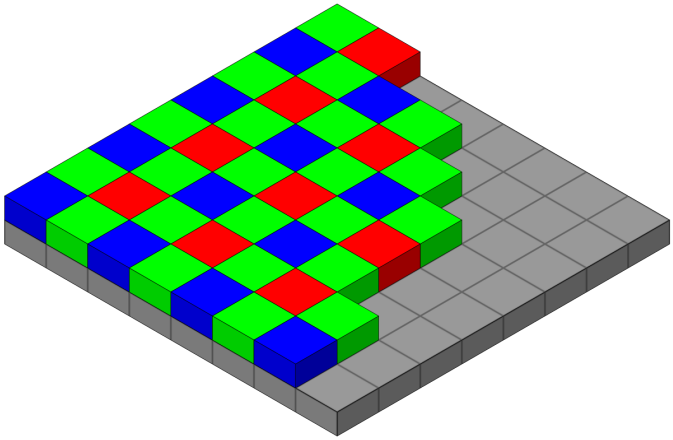


Figure 3.2: A color filter array forming an RGGB Bayer pattern on top of a a camera photosensor array to preserve color information through the use of a single camera sensor.

Each photosensitive element in a sensor with a Bayer CFA is sensitive to one of three different light wavelengths, blocking out light outside of that range. Therefore, each subset of pixels with the same color filter form each channel of the color image. This process is visualized in Figure 3.3. An RGGB Bayer pattern corresponds to a CFA with one red, two green and one blue pixel for each 4 pixel quadrant. This is the most common CFA pattern, but it's not the only one. Other variants exist for different applications, but the RGGB pattern is the most widely adopted one as it tends to produce the most natural-looking color representation.

3.2.2 ISP

All camera based hardware devices make use of an Image Signal Processor (ISP) that can be implemented either as efficient image processing hardware blocks, or purely as programmable software running on traditional compute hardware blocks. The purpose of an ISP is to transform the raw RGGB data from the CFA sensor to a usable image that an end application can make use of.

The most important part of the ISP is the demosaicer. When working with a

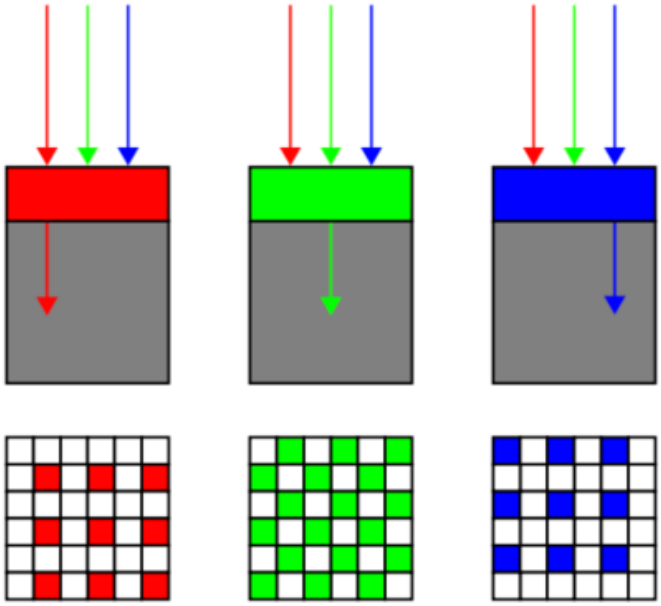


Figure 3.3: A Bayer CFA filter on a sensor causes each photosensitive element of the sensor array to react to a narrow range of light wavelengths, thereby separating at the pixel-level color information in the resulting image.

single sensor with a CFA, it can be seen that the resulting image has sparse color information. Each of the three resulting color channels has pixels that are not necessarily packed next to each other, but rather are spread out through the image through a predetermined strided pattern. Note also that the green channel holds twice as much information as the red and blue channels.

In contrast to the three-sensor system, camera sensors with CFA patterns obviously reduce the image resolution of each of the resulting color patterns. To combat this, the demosaicer implements an interpolation algorithm such as the one that can be seen in Figure 3.4. Demosaicing essentially uses a bilinear interpolation filter to generate the missing color information on pixel locations where each channel lacks any data. By integrating information from the surrounding pixels, the missing information can be re-generated through this interpolation, resulting in a three color channel image where each channel has the full original resolution of the image sensor.

The ISP is also responsible for a series of other image manipulation operations,

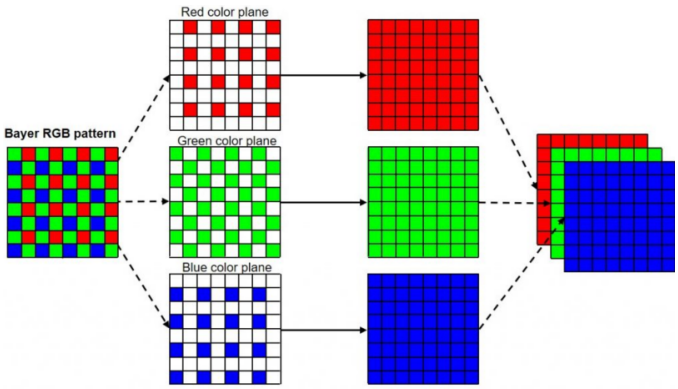


Figure 3.4: A demosaicing algorithm helps to fill in the missing color data for each channel on pixels where no data is available as a result of the CFA color sparsity.

such as color correction, denoising, sharpening, gamma correction, defective pixel correction, among many others. Different hardware systems will have different ISP pipelines that perform different operations, according to the characteristics of the image sensor and the intended usage of the final produced images.

However, something that is very common among all ISP pipelines is a color space transformation to and from RGB into other color spaces such as YUV. This is because many of the operations that an ISP pipeline performs can be better implemented when they work in this YUV color space. Therefore, the ISP often maintains a *copy* of the image data in the YUV color space, making YUV one of the possible output formats for most ISPs. One particular but very common use case for this is in video recording, as many video encoders and decoders can work more efficiently in this color space. Therefore, instead of obtaining an RGB image from the ISP and converting it to YUV in software, it is often possible to directly request YUV data from the ISP instead.

3.2.3 YUV Color Space

Historically, the RGB color space is the preferred color model when dealing with digital imaging. The reason being its compatibility with other systems and easy visualization. Additionally, the reason for colors to be digitally represented in this manner is primarily biological in nature, as the human visual system is trichro-

matic and composed of three types of cone cells – each one sensitive to a different wavelength of light, roughly corresponding to red, green and blue. As a result, most screen displays are composed of pixels of these three colors such as to better activate the light receptor cones in the human retina, and to generate a signal that can display an image on such screens, information for these three color channels needs to be given separately. Needless to say then, RGB images are composed of three channels, one for each of these same three component colors. Image intensity is given by the common increase of the values in all three color channels simultaneously, whereas chromatic information comes from the variation in values of the three channels for every pixel in a photograph. This leads to relatively large amount of redundancy in the input data, as all three channels will usually have a high level of correlation between them when the saturation of the image is low.

Given that these three color channels are all as important as one another, they need to be treated with almost equal importance by the neural network – but this leads to wasted resources spent in extracting redundantly similar features for each data channel separately. The question arises if there is a better way to represent the image data such as to help with the neural network's learning process, making as efficient use as possible of the given data distribution. As usual, we look to the biological visual system for inspiration on how to better treat this information. It is well known that the human visual cortex is much more sensitive to changes in luminance than to changes in color [42]. It is for this reason, that image recognition with purely grayscale representations of images is actually possible, albeit with lower accuracy than with full color information. Therefore, it would be beneficial to have a system where the intensity can be treated with the attention it demands, while still making use of chromatic data, but only as supporting information.

There exist various color models which follow such a structure. HSL being one of the better known ones, where the three data channels represent hue (H), saturation (S) and luminosity (L). Similarly the Lab color space uses one channel for lightness (L), and two channels for opponent color information (a and b). Finally, another well known system is the YUV color model, where the image intensity is represented in one channel (Y) and the color information is given in two supporting channels (U and V). An example of an image represented in this colorspace can be seen in Figure 3.5. Although all three of these color models have very different specifications, it can be expected for the training process to proceed with roughly the same efficiency using either of these. However, the YUV space is chosen for the system presented in this work for two particular reasons: (i) it is readily available

in the ISP as this color space is used for many other image processing algorithms, and (ii) it can be computed from an RGB image via a simple and efficient linear transformation, therefore it doesn't introduce any biases or non-linear factors.

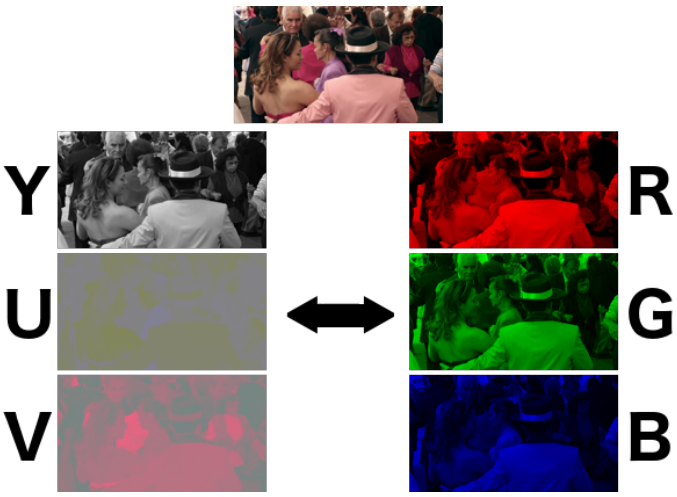


Figure 3.5: A visualization of the YUV and RGB color spaces, where the Y data channel corresponds to luminosity information and the two U and V planes are two opposing chromacity channels.

$$\begin{bmatrix} Y \\ U \\ V \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ -0.147 & -0.289 & 0.436 \\ 0.615 & -0.515 & -0.100 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix} \tag{3.1}$$

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1.14 \\ 1 & -0.395 & -0.580 \\ 1 & 2.03 & 0 \end{bmatrix} \begin{bmatrix} Y \\ U \\ V \end{bmatrix} \tag{3.2}$$

The transformation between these two color spaces is therefore extremely efficient and can be implemented as a series of element-wise MAC computations for all pixels in the image.

3.2.4 Color Compression

Within YUV color formats there exist various ways to transfer the color information. As the luminosity channel carries proportionately far more information than the chroma channels, it is possible to compress the image size by reducing the resolution of the chroma channel planes, while keeping the luminosity channel intact. The reason behind this is that the human visual system is far more sensitive to changes in intensity information than changes in hue. As a result, it is possible to maintain almost the same visual quality (as perceived by human vision) of an image when performing this compression of the chroma channels and reconstructing back the full color image.

This compression becomes an important part of image processing and transfer systems, and as can be expected, there are various standards that define how this compression happens and by what factor the chroma data is downsampled. This is usually represented in an $N : M : L$ format, where N represents how many pixels, out of every four, carry their original luminosity information. M and L then represent the same but for each of the two chroma channels respectively. A visualization of this compression can be seen in Figure 3.6.

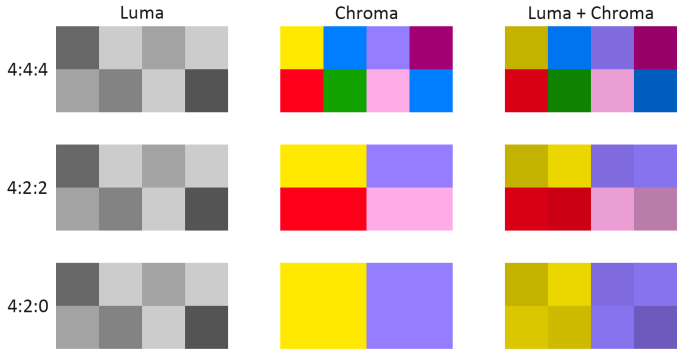


Figure 3.6: Three different compression levels for YUV color information, ranging from full information at every pixel (4:4:4, also referred to as YUV444), down to a single chroma pixel for every four luminosity pixels (4:2:0, also known as YUV420).

The compression levels of 4:4:4 and 4:2:0 result in chroma channels that maintain the original aspect ratio of the image, while a compression level of 4:2:2 results in chroma channels with an aspect ratio twice as wide as the original image, due to

the asymmetric dropping of color information pixels in both dimensions. Figure 3.7 shows the effect of the different compression levels on image quality. There also exist other formats that allow for even further compression but they are no longer useful for image processing as too much information is lost. For the purposes of this work, we choose 4:2:0 (commonly referred to as YUV420) as this level allows data compression, while maintaining spatial proportions correctly without requiring additional interpolation or resizing.

Figure 3.7: The effects of the three different compression levels and how they impact the visual quality of the image.

The next question comes on how the image data is actually packed on a camera byte stream. This is dependent on different hardware criteria related to the camera sensor and the device ISP. Usually, these formats are standardized across a range of devices, and it will depend on hardware vendors which format(s) are available for consumption by the image processing software. In the case of Android phones, however, a very common image data format is NV21, which is best described graphically in Figure 3.8.

This particular format is useful in that the order in which data is packed allows for a very efficient method for extracting the image channel information. Note: In accordance to video standards, most hardware cameras output image data in YUV format, but a slight distinction is that digital cameras actually use the YC_bC_r format, which is the digital equivalent of the YUV analog format specification, where the difference between the two spaces simply reflects some minor variations in the transformation coefficients to correct for some common biases that result from digital signaling. However, it is customary to refer to this format simply as YUV as well, so following convention, the YUV term will be used here.

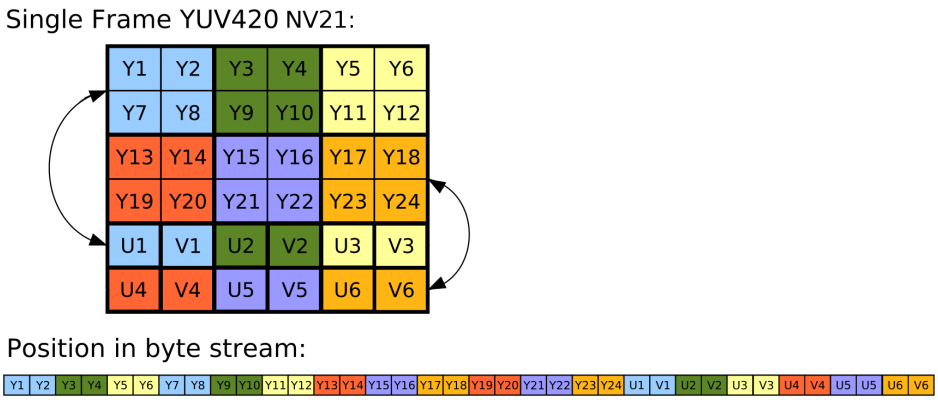


Figure 3.8: The NV21 image format which packs a YUV420 chroma-subsampled image into a byte stream consisting of contiguous luma values, followed by interleaved chroma values.

3.2.5 Contrast Normalization

Computer vision neural networks analyze input images to produce their results. As with any machine learning algorithm, it is extremely important to pre-process the input data in a manner that leads to better analysis by the system.

Prior to processing the image data by the first layer of neurons, the luminosity channel of the image is normalized. The purpose of this is to make a preemptive attempt at levelling out the effects caused by illumination differences. There exist a wide variety of changes that an image can undergo due to varying conditions at the time it is captured. The idea is to normalize the intensity value variations due to light, while keeping the intrinsic changes that are due to the object's own shape. After all, exposure and brightness information has lower importance than shape and edge information when determining the identity of an object.

As can be expected, there are various methods for accomplishing this task, with varying degrees of success. The correct choice of pre-processor to use will vary for each application in particular, but regardless of the chosen algorithm, this is always an important step to take, as with any other machine learning task demanding the normalization of data prior to processing. In other machine learning algorithms, this process is sometimes known as feature scaling, a reference to the fact that it multiplies the data by a factor which globally spreads it out to better cover over a

predefined interval – normally zero-centered.

For this work, however, contrast normalization is proposed as a method that can better accentuate the features that are most important to an image. The main reason being that this system is biologically inspired [43, 44] usually outperforming other alternative methods. As opposed to many common pre-processing algorithms, contrast normalization is non-linear, which allows for the introduction of better features into the input data.

This process is primarily calculated through image convolution operations, making it somewhat more expensive computationally than other statistics-based algorithms. However, the feature enhancement provided by this algorithm compensates its computation cost by effectively reducing the amount of operations that the neural network backbone must perform to extract very similar features. The effect provided by contrast normalization can be seen in Figure 3.9.

Figure 3.9: An example of the effect that contrast normalization has on an image, where the output values have been remapped to a [0, 255] range for visualization purposes.

Contrast normalization has the effect of enhancing edges and salient features in the image. Edges in an image are mainly the result of a sudden change in value, either from low to high or high to low. These sudden gradient changes are translated to positive or negative values by the contrast normalization process, while areas of constant value or with only small and gradual change tend towards zero.

It is important to note that this normalization is applied on an image in YUV color space with 4:2:0 chroma compression, but only on the luminosity channel. The color channels are not modified in any way, as shifting any of their values would have the effect of remapping image color hues significantly which would only lead to confusion in the neural network, something which is not desirable

when the chromatic value of specific regions in the images have great significance attached vital for accurate classification. Furthermore, due to the nature of the YUV color space, the typical values of U and V chroma channels are already zero centered, and range within an acceptable [-1, 1] value distribution.

The contrast normalization filter is achieved by applying convolution with a Gaussian kernel. As Gaussian kernels are fully separable, this can be applied much more efficiently via the separable convolution method [45]. This basically applies two sets of 1D Gaussian convolutions on the input data, one horizontally and one vertically. The 1D convolution kernel used is a single row Gaussian vector. This operation is considerably less computationally expensive than a full 2D kernel. That is, separable convolution has a complexity of $\mathcal{O}(2n)$, compared to $\mathcal{O}(n^2)$ in the case of the full 2D Gaussian kernel.

The contrast normalization filter is performed by the following set of operations:

$$C^S(I,K) = I - \left(K^T \otimes (K \otimes I)\right) \tag{3.3}$$

$$C^D(I,K) = \frac{I}{\sqrt{K^T \otimes (K \otimes (I \circ I))}} \tag{3.4}$$

$$C^N(I,K) = C^D\left(C^S(I,K), K\right) \tag{3.5}$$

Where I is the input image at each operation, usually modified with a 1D Gaussian kernel vector K and its corresponding transposed kernel K^T to perform a convolution in both the horizontal and vertical directions of the image.

The $C^S(I,K)$ function is a subtractive operation, where the dual convolutions have the effect of generating a localized mean representation of the input image, thus, subtracting this mean from the original input image yields a zero-centered normalized image.

The $C^D(I,K)$ function is a divisive operation, where the dual convolutions operate on the input image squared (element-wise matrix multiplication by itself). The square root of this amount represents a localized standard deviation of the input image, thus, dividing the input by its own standard deviation results in a normal distribution with $\sigma = 1$.

The final $C^N(I,K)$ function to implement the contrast normalization operation, then, is that of the previous two operations applied sequentially with the same ker-

nel, resulting in a normalized image with values centered around 0, and a standard deviation of 1 – priming it for a machine learning algorithm to process. The localized convolutions applied throughout the process also have the effect of flattening gradients over the image, while enhancing the ever important edge outlines describing the object contours, both of which are ideal for easier recognition. Figure 3.10 shows this effect on the distribution of values of the input image luminosity data, and the output of the algorithm on that same data distribution.

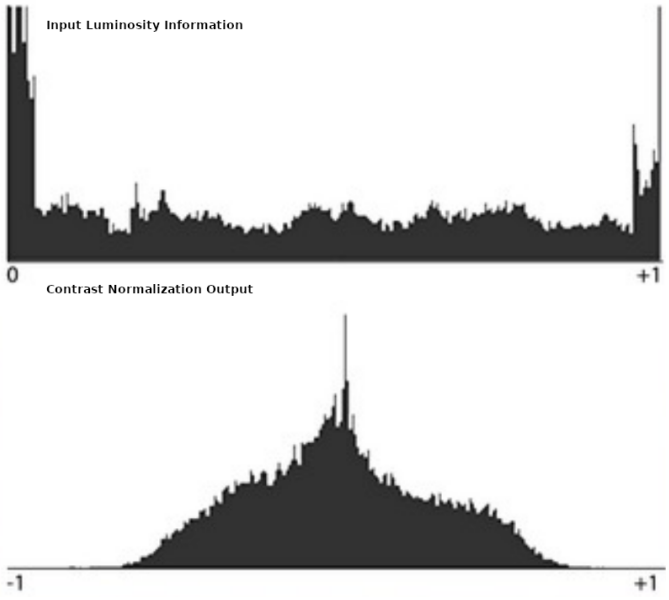


Figure 3.10: **Top:** A histogram of the value distribution of the input luminosity channel for the image in Figure 3.9, which can be seen to fall within the [0, 1] value range, and with peaks at either end. **Bottom:** A histogram of the values of the output of the contrast normalization algorithm applied to that image, where the ranges now fall within a [-1, 1] range, they are zero-centered and following a Gaussian bell shape distribution.

The other important optimization that happens in this operation is striding. All convolution operations are applied with a stride of 2×2, which essentially means that the filter skips every other row and column of the input image. As a result, the output image has a resolution of one half the width and height of the input image. This causes the result of the contrast normalization pre-processor on the luminosity channel to have the exact same size as the chroma subsampled channels

due to the 4:2:0 color compression.

The end result of this operation is three color channels in YUV color space, all with the same resolution which is half of that of the input image size. All three channels will be zero centered and in the range of [-1, 1]. The luminosity channel will further have a standard deviation of 1, and the chroma channels will have unmodified magnitudes that the computer vision system can still use to attribute color information to different regions of the image. This image is now ready to be processed by the neural network.

3.3 Network Architecture

The backbone of the neural network requires an architecture that is better guided towards efficient execution, as neural networks are very compute intensive, this becomes the bottleneck of the entire system. Having an optimized architecture is a delicate balance between runtime speed and prediction accuracy, and therefore requires a good trade-off to be made when targeting an edge application.

3.3.1 Classifier Backbone

The neural network architecture used for this detector system starts with the development of a classifier-like network. This network has the task of taking an input image and predicting an output class label.

The nature of the task, therefore, already dictates the input and output shape constraints: The input shape must match the dimensionality of input images, wile the output shape must be a vector with as many elements as there are target classes.

The network on which our system is based upon is a standard CNN. Figure 3.11 depicts the layer structure of such a network, and it is the reference architecture used throughout this work to describe the concepts of the proposed framework. Note that the depicted architecture represents a standard template which can be used to build larger or smaller versions of the network, by adapting either the size of the input images, the number of layers, or the neurons on each layer. Naturally, such changes will have an impact on the trade-off between accuracy and runtime performance.

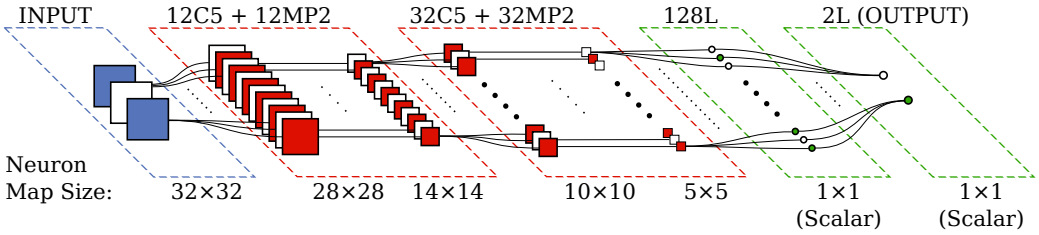


Figure 3.11: A typical convolutional neural network architecture, with three input neurons for each color channel of an analyzed image patch, two feature extraction stages of convolutional and max-pooling layers, and two linear layers to produce a final one-vs-all classification output.

In the initial stages of the CNN, a neuron consists of a two-dimensional grid of independent computing units, each producing an output value. As a result, every neuron will itself output a grid of numerical values, a data structure in $\mathbb{R}^2$ referred to as an activation map. When applying CNNs to image analysis, these maps represent an internal state of the image after being processed through a connective path leading to that particular neuron. Consequently, maps will usually bear a direct positional and feature-wise relationship to the input image space. As data progresses through the network, however, this representation turns more abstract as the dimensionality is reduced. Eventually, these maps are passed through one or more linear classifiers, layers consisting of traditional single unit neurons which output a single value each. For consistency, the outputs of these neurons are treated as 1×1 single pixel image maps, although they are nothing more than scalar values in $\mathbb{R}^0$.

3.3.2 CNN Layers

The first layer in the network consists of the image data to be analyzed, usually composed as the three color channels. The notation $N_j X K_j$ is used to describe all subsequent layers, where N_j is the neuron map count of layer j, $X \in \{\mathcal{C}, \mathcal{MP}, \mathcal{L}\}$ denotes the layer type group (Convolutional, Max-Pooling, and Linear), and K_j is the parameter value for that layer. So for example $12\mathcal{C}5$ describes a convolutional layer of 12 neurons with a kernel size of 5. In this manner, a configuration for the full network can be given by stating the individual layers, where for example the network depicted in Figure 3.11 can be defined as having the configuration:

$$12\text{C5} + 12\text{MP2} + 32\text{C5} + 32\text{MP2} + 128\text{L} + 2\text{L}$$

The network has several feature extraction stages composed of a combination of convolutional and max-pooling layers. The first part of every $\mathcal{C} \rightarrow \mathcal{MP}$ feature extraction stage is the convolutional layer. Here, each neuron linearly combines the convolution of one or more preceding maps. The result is a map slightly smaller than the input size by an amount known as the kernel padding, which arises from the boundary conditions of the valid convolution algorithm. It is defined as ${}^{K_j}/_2 - 1$, where K_j is convolutional kernels size of layer j. Therefore, the layer's map size will be given by $M_j = M_{j-1} - {}^{K_j}/_2 - 1$, where M_{j-1} is the the preceding layer's map size. Note that although not explicitly stated in the network configuration, each convolutional layer is inherently followed by a non-linearity function, such as $tanh(x)$.

A max-pooling neuron acts on a single map from a preceding convolutional neuron, and its task is to subsample a pooled region of size K_j. The result is a map size that is inversely proportional to said parameter by $M_j = {}^{M_{j-1}}/_{K_j}$. The data may then be passed to one or more additional $\mathcal{C} \rightarrow \mathcal{MP}$ feature extractors.

Linear layers classify feature maps extracted on preceding layers through a linear combination similar to a Multi Layer Perceptron (MLP) – always working with scalar values – such that $M_j = 1$ at every layer of this type.

Figure 3.12 shows how information flows through the various layers of the CNN, leading to this classification for a given input image among two possible classes that this particular network was trained for. From left to right, the flow starts with YUV image information. The first convolutional stage consisting of 12 convolutional units producing image-like activation maps of roughly half the size of the input images. This $12C5 + 12MP2$ stage produces twelve activation maps (only the first three are visualized for brevity). This data then flows to the second convolutional stage, a $32C5 + 32MP2$ layer configuration that produces 32 activation maps of even lower resolution.

These two convolutional stages serve as a form of feature extractor that reduce the input image to a lower dimensional representation. By repeated convolutions and max-pooling operations, the data is reduced from a tensor of size $32 \times 32 \times 3 = 3072$ data points, to one of size $5 \times 5 \times 32 = 800$. These 800 data points serve as the input for the final stage of the CNN, a simple MLP which is able to classify the 800 non-linear features extracted from the input image into the final number of target classes the recognizer is trained upon.

The output layer in the linear classifier has as many neurons as there are classes

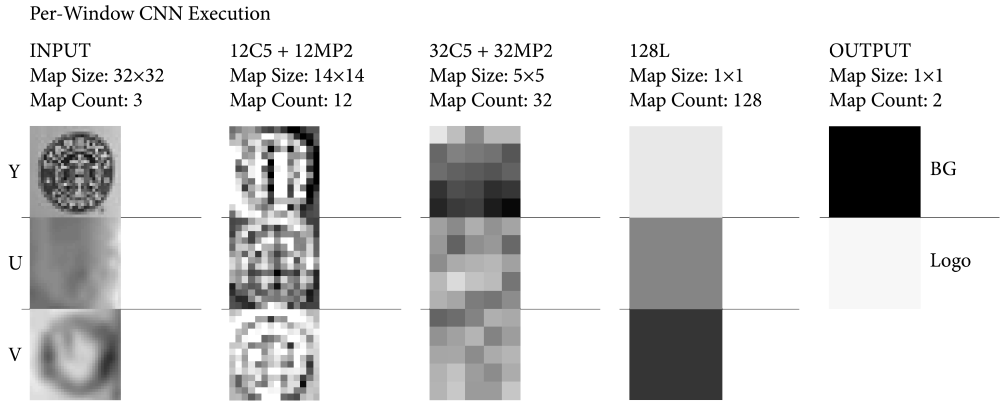


Figure 3.12: Visualization of the first three neuron maps at each stage of the CNN when performing inference on an input image window of size 32×32. Note the data size reduction induced at each stage. The output of this execution consists of two scalar values, each one representing the likelihood that the analyzed input image belongs to that neuron's corresponding class. In this case the logo has been successfully recognized by the higher valued output neuron for class "Logo".

in the problem to solve, plus one. The extra output neuron usually corresponds to a catch-all background class for negative classifications. This layout fully defines the proposed neural network classification backbone.

3.3.3 Connectivity Mapping

The connectivity of the neural network is very important in that prior domain knowledge can be integrated into the system, and in this manner mold the algorithm to better suit a particular goal. Unlike traditional neural networks, neurons in each layer are not simply connected to all neurons in the previous layer, but rather, information bottlenecks can be introduced through selective connectionism that force the influence of data to take specific routes within the network, thereby modifying the behavior of individual neurons in a particular manner. This concept is largely rooted in how specific neural pathways exist within the visual cortex.

As explained in section 3.2.3, more importance needs to be assigned to the luminosity channel of the input image than to the color data channels. Structure-wise, this influence is built into the system through a custom connection map between the first two layers. By doing this, the prior knowledge of asymmetrical channel in-

fluence can be embedded into the network model. A connectivity mapping is thus formed between the input neurons and the first convolutional layer in such a way that the Y channel input neuron fully connects to proportionately larger subset of the convolutional neurons in the first layer, whereas each of the U and V channel input neurons connect to only smaller subsets of neurons each. Assigning more connections to certain neuron greatly increases the effect its stimulus information has over the entire system state. It is important to note here that each channel will be processed separately on this first convolutional layer of the CNN, and it is not until the next feature extractor stage where the information from all channels is finally mixed together. The final classifier backbone architecture implementing this connectivity mapping can be seen in Figure 3.13.

In the next layer, each of the non-linear max-pooling neurons is directly connected to exactly one convolutional neuron in the preceding layer. This is merely due to the fact that these neurons are meant to operate only over a single image, therefore taking only one input each. The activated output of these layers, however, connects to the next stage of the CNN in a more intricate manner.

This next stage of the network consists of another convolutional layer. As the convolution operation is one of the most expensive operations throughout the network, the connectivity map between these two layers is modified so as to reduce the total number of operations needed to be performed, especially since this is the most computationally demanding layer of the CNN structure. Specifically, 12 neurons in the first convolutional layer fully connected to 32 neurons in the following stage would require $12 \times 32 = 384$ convolution operations. Keeping in mind the recursive nature of the convolution algorithm, this seemingly low number of convolutions actually represents 1.9 million arithmetic operations, not taking into account the memory lookup calculations performed by the algorithm.

On the other hand, if a sparse connectivity map is used between these two stages, the required number of operations can be considerably reduced – by implementing a 1/3 random connection map, for example. That is, each neuron in the second stage connects only to 1/3 of the neurons in the previous layer – thereby decreasing the amount of convolutions performed proportionally to the map reduction factor. In this case, only $4 \times 32 = 128$ operations are needed, exactly one third as many as in the fully connected case.

This process reduce the computational cost of the network, but another way of looking at this is by considering that given a fixed number of computations, it is

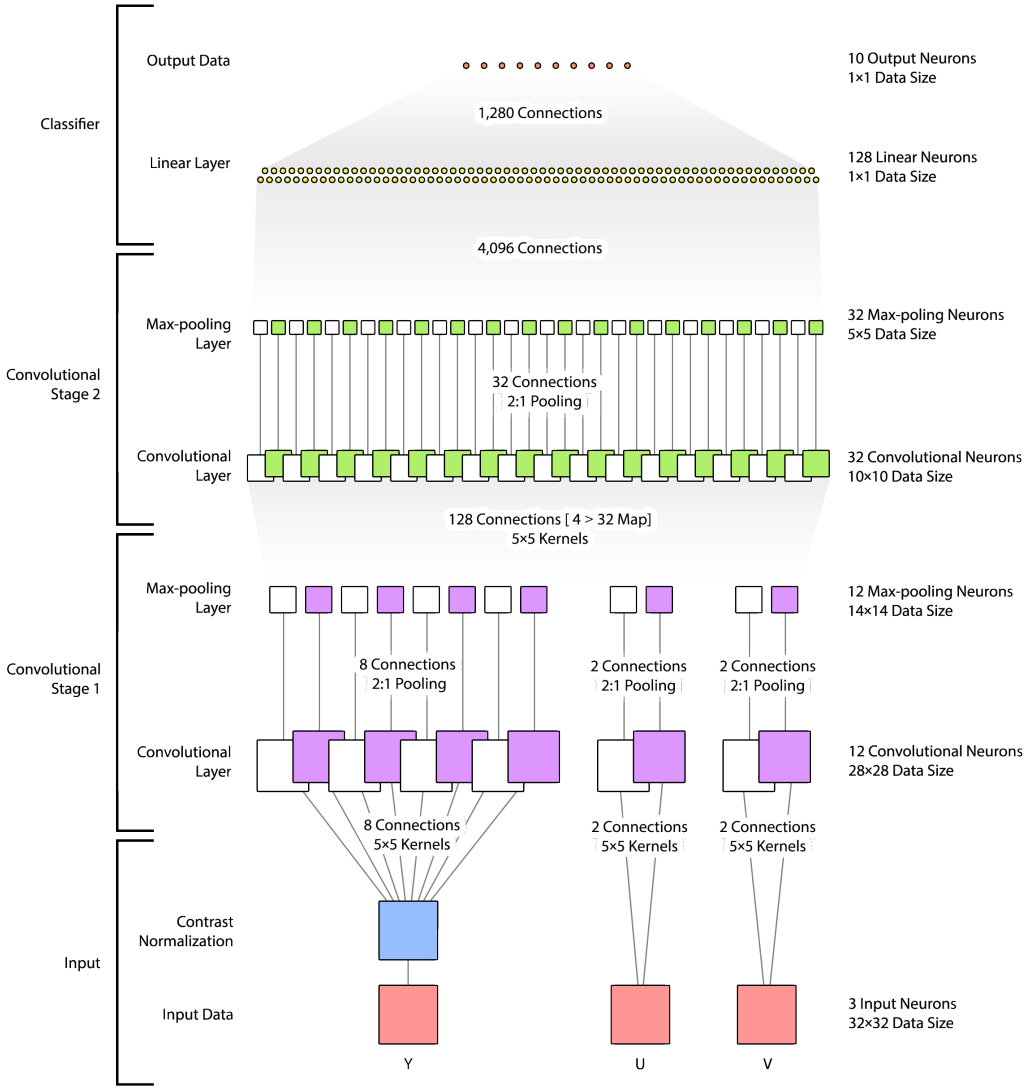


Figure 3.13: The proposed convolutional neural network backbone architecture with a *12C5 + MP2 + 32C5 + MP2 + 128L + 10L* configuration, but with a bottleneck in the connectivity mapping between the YUV image information and the first convolutional layer, and again between the convolutional feature extraction stages 1 and 2, where a $4 \rightarrow 32$ connectivity map is used, thus generating only 128 neural connections.

actually more beneficial for the network to learn more unique data structures to have more neurons with sparse connectivity than fewer neurons with dense connectivity. This sparse connectivity can be overcome by the Backpropagation algorithm as it will forcefully learn the most optimal use of the available connections and each neuron will learn the best possible kernel weights for the few connections it has access to.

In larger neural networks, this procedure is often achieved by different methods of network pruning [46, 47]. Pruning is often done as an iterative task, where a neural network is first trained with full dense connections, then the connections that are activated the least are selectively removed, after which, the network is trained again to make the model adapt to the new connectivity. This process is often repeated for a few iterations until some convergence criteria is reached. However, such methods applied on extremely small neural networks such as the one proposed by this work do not yield good results due to codependent neural pathways. Therefore, forcing a sparse connectivity at the design stage allows training the model in one single pass with this constraint already built in, and as such, it leads to better learning for such network architectures.

3.4 Neural Spatial Extension

To solve the task of object detection, it is necessary to find not only *what* an object is, but also where in the image it is located. The problem of object detection therefore can be divided into two parts: (i) classification, and (ii) localization. And this must be done for one or more objects that may appear in the image.

As expected, the task of classification is partially solved by the classifier described in Section 3.3. However, a classifier classifies a full image as belonging to one class or another, and is not capable of producing any information related to localization within the image.

Therefore, the inference methodology must be adapted to provide contextual information according to the position of objects within the input image space. State of the art object detection systems, such as SSD [35] resolve this problem by adding an extra *head* to the network, which acts as a regression layer to predict a continuous numerical value for each of the four coordinates that define a bounding box. While extremely effective and accurate, this process obviously places additional computational requirements on the neural network implementation.

This section describes an alternative method by which localization information can be extracted by extending the spatial inference space of the network. This information can then be further analyzed to produce the final bounding box locations to localize objects in the input image.

3.4.1 Sliding Window

The proposed concept of shared maps is rooted on the traditional approach of the sliding window. A sliding window is a brute force methodology by which inference over a large image can be performed by moving a window throughout the entire image and classifying each region through a computer vision model such as a CNN in this case. This form of brute force search basically goes through the whole image to find where within the image something of interest might be located.

This sliding window approach is mainly defined by two quantities, the window size S, usually fixed to match the CNN's designed input size; and the window stride T, which specifies the distance at which consecutive windows are spaced apart. This stride distance establishes the total number of windows analyzed W for a given input image. For an image of size $I_w \times I_h$, the window count is given by:

$$W = \left(\frac{I_w - S}{T} + 1\right)\left(\frac{I_h - S}{T} + 1\right) \quad \Longrightarrow \quad W \propto \frac{I_w I_h}{T^2} \tag{3.6}$$

Figure 3.14 shows this method applied on an input image downsampled to 144×92, extracting windows of $S = 32$ for the simple case where $T = S/2$. A network analyzing this image would require 40 executions to fully analyze all extracted windows. The computational requirement is further compounded when a smaller stride is selected – an action necessary to improve the resolving power of the classifier: at $T = S/8$, 464 separate CNN executions would be required.

3.4.2 Shared Maps

The method proposed introduces a system where the stride between regions of interest has no significant impact on the execution time of the $\mathcal{C} \rightarrow \mathcal{MP}$ feature extraction stages, as long as the selected stride is among a constrained set of possible values. This is achieved by allowing layers to process the full image as a single shared map instead of individual windows. Constraints in the possible

Figure 3.14: An overview of the sliding window method, where an input image is subdivided into smaller overlapping image patches, each being individually analyzed by a CNN. A classification result is then obtained for each individual window.

stride values will result in pixel calculations to be correctly aligned throughout the layers, a critical requirement for this system to perform inference correctly.

CNNs have a built-in positional tolerance due to the reuse of the same convolutional kernels over the entire neuron map. As a result of this behavior, their output is independent of any pixel offset within the map, such that overlapping windows will share convolved values. This is demonstrated in Fig. 3.15.

This leads to the possibility of streamlining the feature extractors by running their algorithms over the full input image at once. Hence, each $\mathcal{C} \rightarrow \mathcal{MP}$ neuron will output a single map shared among all windows. This greatly reduces the expense of calculating again convolutions on overlapping regions of each window. Figure 3.16 shows an overview of the shared map process, which passes the input image in its entirety through each stage of the network.

By doing this, the output layer now produces a continuous and localized class distribution over the image space, a result which contrasts greatly to that of a single classification value as was previously seen in Fig. 3.12. An account of the window size and stride is also displayed, illustrating how it evolves after each layer,

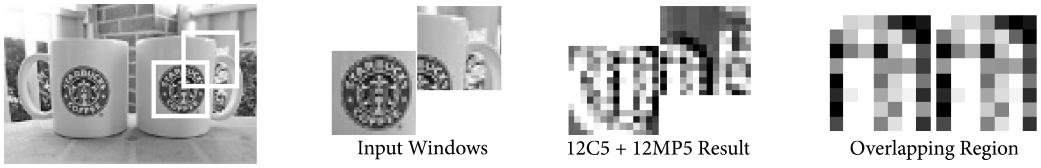


Figure 3.15: Two adjacent windows extracted from an input image, passed through the 12C5 + 12MP5 feature extractor. A detailed view of the convolved maps in the overlapping top-right and bottom-left quarters of each window shows that these areas fully match.

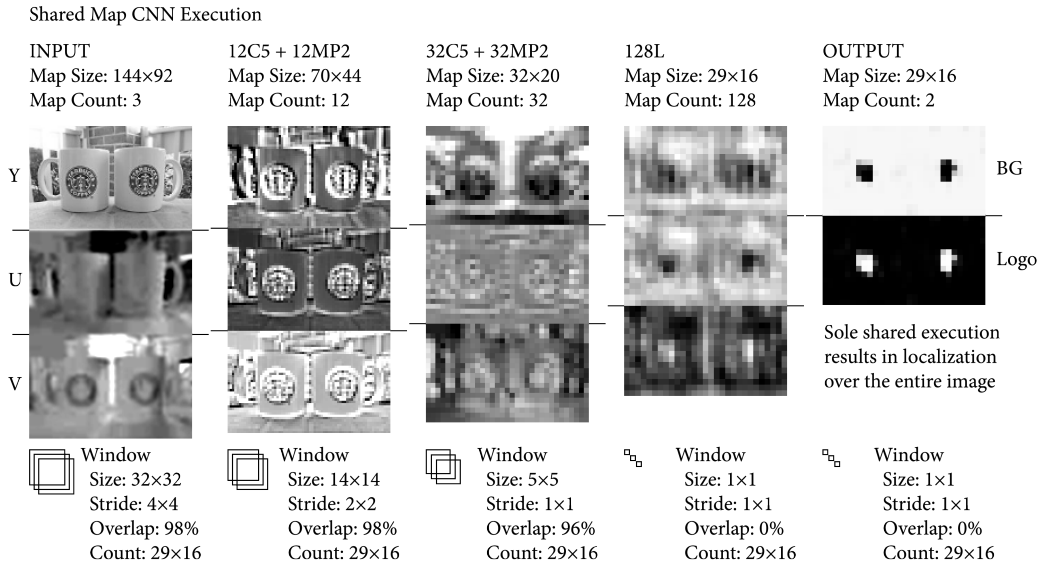


Figure 3.16: The shared map execution method for a convolutional neural network, where each layer processes an entire image in a single pass, and each neuron is now able to process maps with dimensions that far exceed the layer's designed input size.

while the total window count remains the same. Here, the correspondence of each 32×32 window in the input image can be traced to each one of the pixels in the output maps.

More importantly, however, the output of this execution consists of image maps where each pixel yields the relative position of all simultaneously classified windows. Similar to the per-window execution method, the intensity value of a pixel in the output map represents the classification likelihood of the corresponding window. Note how the relative position of the logo in the input image has been discovered after only one shared map execution of the network. It is this positional context that provides the basis for localization, and as such it turns a classification network performing inference on a single image patch to an object detection problem that performs inference over a larger image space, with results encoding information to locate the relevant objects.

3.4.3 Stride Configuration

The operation of the shared map process relies greatly on the details of the dimensionality reduction occurring at each layer within the network. For this reason, it is necessary to lay certain constraints that must be enforced when choosing the optimum sliding window stride.

At each layer, the window size and stride are reduced until they eventually become single pixel values at the final linear layers. The amount of reduction at each stage varies according to the type of the layer and its parameters. All of these quantities can be found in a well defined manner as given by:

$$S_j = \begin{cases} S_{j-1} - K_j - 1 & if \;\; j \in \mathcal{C} \\ S_{j-1}/K_j & if \;\; j \in \mathcal{MP} \\ S_{j-1} & if \;\; j \in \mathcal{L} \end{cases} \tag{3.7}$$

$$T_j = \begin{cases} T_{j-1} & if \;\; j \in \mathcal{C} \cup \mathcal{L} \\ T_{j-1}/K_j & if \;\; j \in \mathcal{MP} \end{cases} \tag{3.8}$$

Where the window size S_j and its stride T_j at layer j depends on the various parameters K_j of the layer and the window size and stride values at the preceding $j-1$ layer. This equation set can be applied over the total number of layers of the network, while keeping as the target constraint that the final size and stride must remain whole integer values. By regressing these calculations back to the input layer $j = 0$, one can find that the single remaining constraint at that layer is given by:

$$T_0 \equiv 0 \bmod \prod_{j \in \mathcal{MP}} K_j \tag{3.9}$$

In other words, the input window stride must be perfectly divisible by the product of the pooling size of all max-pooling layers in the network. Choosing the initial window stride in this manner, will ensure that every pixel in the final output map is correctly aligned throughout all shared maps and corresponds to exactly one input window. Fig. 3.17 follows the evolution of the window image data along the various layers of the sample network architecture, showing this pixel alignment throughout the CNN.

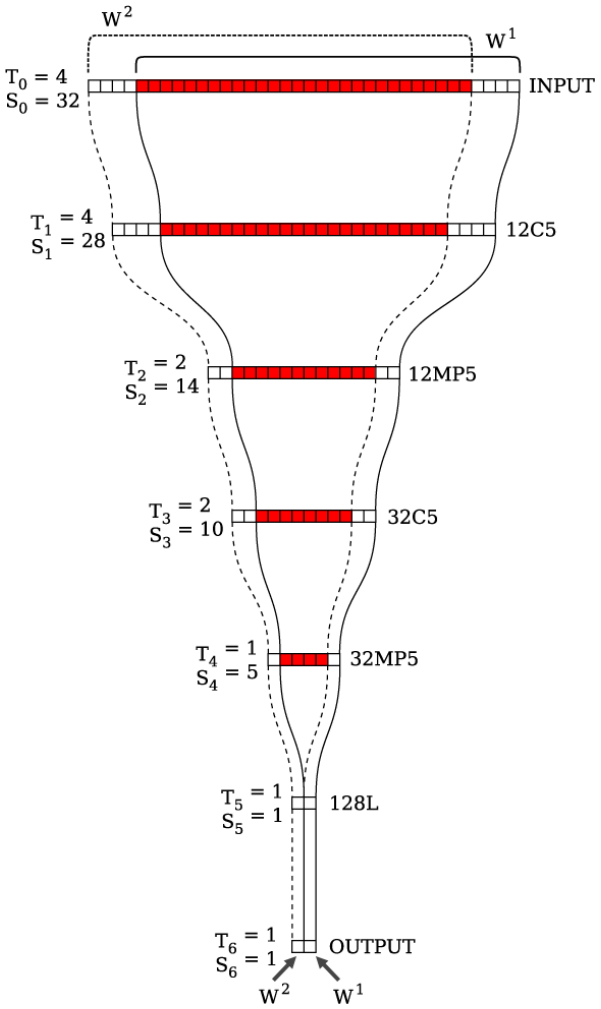


Figure 3.17: The CNN layers and their effect on the window pixel space, illustrated in one dimension for simplicity. Two successive 32×32 windows W^1 and W^2 are shown. Overlapping pixels at each layer are shaded. Starting with an input layer window stride $T_0 = 4$, the final output layer results in a packed $T_6 = 1$ window stride, so that each output map pixel corresponds to a positional shift of 4 pixels in the input windows, a relationship depicted by the darkened column path traversing all layers.

3.5 Discrete Inference of CNN Output

The output from the convolutional neural network as seen in Figure 3.16 consists of multiple individual maps, where each one gives a visual depiction of the relative confidence, per class, that the network predicts for every window that has been sampled.

The common practice to obtain a final classification from such an output value set is to identify which class has a higher output value from the CNN for each each sampled window (here, each pixel in the output map). While efficient, results from this procedure are not always ideal because they only take into account windows separately and interactions between contiguous windows are ignored.

Furthermore, maximum value inference is prone to false positives over the full image area. Due to their non-exact nature, neural network accuracy can decrease by finding patterns in random stimuli which eventually trigger neurons in the final classification layer.

In the previous example, the output classification maps were very clear and they easily define where the objects of interest are located. However, in more cluttered images, the classification maps may not be as confident and can produce much noisier outputs. Such an example can be seen in Figure 3.18 where there exists some confusion in the classification results and some *blobs* in these output maps seem to indicate falsely detected objects. However, such occurrences tend to appear in isolation around other successfully classified image regions. It is therefore possible to improve the performance of the classifier by taking into account these nearby classification windows.

There exist many statistical approaches in which this can be implemented, such as (i) influencing the value of each window by a weighted average of neighboring windows, or (ii) boosting output values by the presence of similarly classified windows in the surrounding area. However, we propose discrete energy minimization through belief propagation [48] as a more general method to determine the final classification within a set of CNN output maps. The main reason being that graphical models are more flexible in adapting to image conditions and can usually converge on a globally optimal solution.

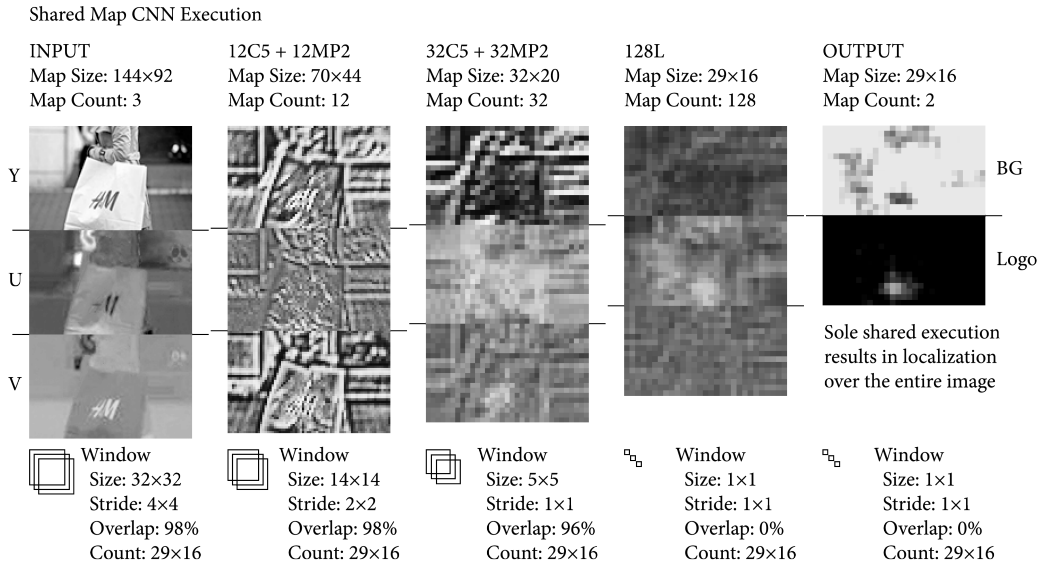


Figure 3.18: The shared map execution method for a convolutional neural network, where the output classification maps are less confident and produce somewhat noisier results.

3.5.1 Pairwise Markov Random Field Model

Images can be treated as an undirected cyclical graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$, where nodes $n_i \in \mathcal{N}$ represent an entity such as a pixel in the image, and graph edges $e_{ij} \in \mathcal{E}$ represent the relationship between these nodes. If, for simplicity, 4-connectivity is used to represent the relationship between successive nodes in a graph; then each node will be connected to four others corresponding to its neighbors above, below, and to each side of the current element.

The output space of the convolutional neural network can therefore be represented in this manner through a graph. However, instead of describing pixel intensity values, each node in the graph represents the classification state of the corresponding window. This state takes on a discrete value among a set of class labels $c \in \mathcal{C} \equiv \{BG, Logo\}$ corresponding to the classification targets of the CNN. Thus, each node in the graph can take on one of several discrete values, expressing the predicted class of the window that the node represents. Figure 3.19 (Left) displays the structure of such a graph.

It can be seen that if nodes represent classification outcomes, there is a strong relationship between them. The reason is that continuity throughout a map tends

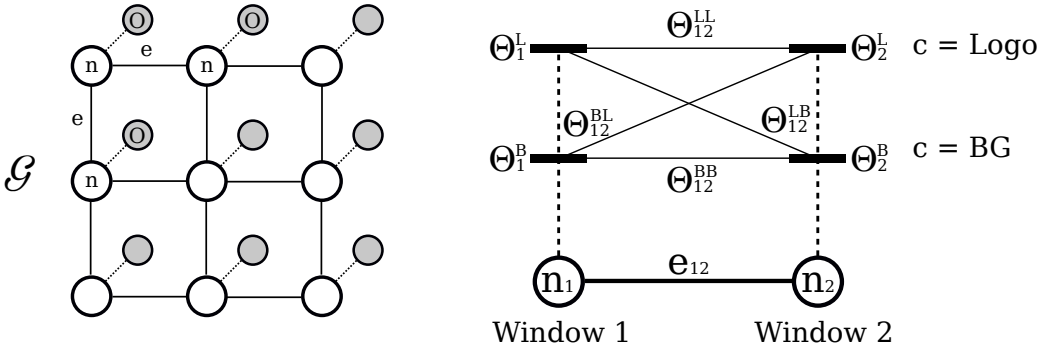


Figure 3.19: **Left:** A subset of the MRF graph $\mathcal{G}$ formed by the CNN output space, where each node n_i represents the classification state of a corresponding window analyzed with the network, whose outputs are implemented into this system as the observed hidden variables O. Nodes have a 4-connectivity relationship with each other represented by the edges e_{ij} thus forming a grid-like cyclical graph. **Right:** A detail of the potential energies assigned to each of two nodes $\{n_1, n_2\}$ connected by edge e_{12}. The singleton potentials $\Theta\ {}_i^a$ correspond to the energy associated with node i if assigned to class a, and the pairwise potentials $\Theta\ ij^{ab}$ are the changes in energy that occur by assigning class a to node n_i and class b to node n_j.

to be preserved over neighboring regions due to strong local correlation in in input images. This inflicts a Markovian property in the graph nodes where there is a dependency between successive nodes. Therefore, this graph follows the same structure as an MRF, and any operations available to this kind of structure will be likewise applicable to the output map.

3.5.2 Energy Allocation

To implement energy minimization on an MRF, it is necessary to assign energy potentials to each node and edge. These energies are usually adapted from observed variables, and in this case, they correspond to the values of the output maps and combinations thereof. Therefore, MRF optimization over a graph $\mathcal{G}$ can be carried out by minimizing its Markov random energy E, given by:

$$E(\mathcal{G}) = E(\mathcal{N}, \mathcal{E}) = \sum_{n_i \in \mathcal{N}} \Theta_i(n_i) + \sum_{e_{ij} \in \mathcal{E}} \Theta_{ij}(e_{ij}) \tag{3.10}$$

Here, $\Theta_i(\cdot)$ corresponds to the singleton energy potential of node n_i, and $\Theta_{ij}(\cdot)$ is a pairwise potential between nodes n_i and n_j. Starting from the CNN output map observations, the singleton potentials can be assigned as:

$$\Theta_i = \begin{bmatrix} \omega_i^{\,0} \\ \omega_i^{\,1} \\ \vdots \\ \omega_i^{\,C} \end{bmatrix} \tag{3.11}$$

$$\omega_i^{\,a} = \sum_{c \in \mathcal{C}} \begin{cases} 1 - (O_i^{\,c})^2 & if\ a = c \\ (O_i^{\,c})^2 & otherwise \end{cases} \tag{3.12}$$

Where C is the total number of classes in set $\mathcal{C}$ (2 in the sample CNN architecture), and $O_i^{\,c}$ is the observed CNN value for window $n_i \in \mathcal{N}$ and class $c \in \mathcal{C}$. In this manner, each ω_i^a value is an MSE-like metric that measures how far off from ideal training target values did the CNN classify window n_i as. Thus, a lower potential value will be assigned to the most likely class, while a higher potential value will be given to other possible classes at this node.

Pairwise potentials can be defined as:

$$\Theta_{ij} = \begin{bmatrix} \delta\,{}^{00}_{ij} & \delta\,{}^{01}_{ij} & \dots & \delta\,{}^{0C}_{ij} \\ \delta\,{}^{10}_{ij} & \delta\,{}^{11}_{ij} & & \delta\,{}^{1C}_{ij} \\ \vdots & & \ddots & \\ \delta\,{}^{C0}_{ij} & \delta\,{}^{C1}_{ij} & & \delta\,{}^{CC}_{ij} \end{bmatrix} \tag{3.13}$$

$$\delta\,{}^{ab}_{ij} = \mid O\,{}^{a}_{i} - O\,{}^{b}_{j} \mid \tag{3.14}$$

Where each value δ_{ij}^{ab} is a straightforward distance metric that measures the *jump* in CNN output values when switching from class a to class b between windows n_i and n_j. Thus, these potentials will be small if the same class is assigned to both nodes, and large otherwise. Fig. 3.19 (Right) shows all energy assignments per node pair.

It is worth noting that these Θ_{ij} pairwise potentials between neighboring windows are the only feature that sets apart this process from the traditional winner-takes-all approach.

3.5.3 Energy Minimization by Belief Propagation

Applying an efficient implementation of Belief Propagation to find the lowest possible energy state of the graph will now yield an equilibrium of class assignments throughout the image output space.

Due to the cycles inherent of image-bound graphs, a special variation of the algorithm must be used, in this case Loopy Belief Propagation [49]. This variation requires the minimization to be run several times until the solution converges and an equilibrium is found. However, due to various existing optimizations for this algorithm, this process is very straightforward and can be solved in polynomial time.

Qualitative results of this algorithms can be seen in Figure 3.20, where a comparison is visualized between the proposed method and the traditional method of independently finding the maximum confidence for each classification window.

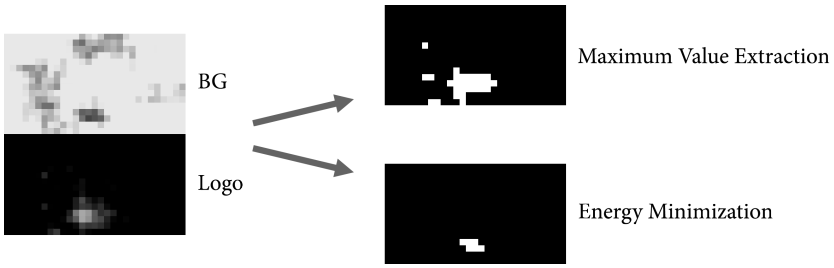


Figure 3.20: Comparison of the final "Logo" classification and localization, applying the classical maximum value per class extraction vs. our proposed energy minimization inference method on the two CNN output maps introduced in Figure 3.18.

3.6 Robustness by Ensembles

Training a miniature neural network such as the one described here, and expecting it to be capable to properly detect complex objects and distinguish them from background clutter and noise of real-world imagery can be extremely naive.

In practice, a classification neural network of this type will be quite capable at properly recognizing the large majority of samples that are presented to it. Thus, when tested against a set of cut-out sample images to perform classification, its true positive inference performance will be quite good. However, it will be prone to make many mistakes when presented with background images or noise, and this becomes apparent when dealing with the larger problem of object detection.

The network is trained with a background class to help it learn the difference between an ear and background noise, but no matter how the training for this class is prepared, a CNN of this size will always be prone to false detections simply due to the internal functionality of neural networks. There will always be patterns or combination of features that can be easily found on natural imagery which will randomly trigger internal neural pathways and thus produce a large false positive rate as well—this could be though of as a type of artificial pareidolia.

To combat this problem, there are two possible solutions. The most obvious answer would seem to be design and train a larger network architecture. While effective, and capable to reduce the number of false positives, there are diminishing results with having a single heavy architecture attempt to resolves this task.

The alternative solution is to create an ensemble of neural networks consisting

Figure 3.21: The three scales that are used for every data sample in the training dataset to train a 3-CNN ensemble system.

of multiple classifiers, each one different in a specific manner. Normally, these ensembles all analyze the same data input, and their different outputs are then combined to create a final result whose accuracy will usually be larger than that of any single classifier running by itself [20].

A variation on this idea is proposed here, in that the ensemble of networks does not process the same data, but rather different data is presented to each member of the ensemble. Therefore, each of the classifiers must then be trained to specialize in the kind of data which will be presented to it, and it will become an expert for that particular preparation of the data.

The main idea then is to feed to three neural networks three different images, each one corresponding to the same image region being analyzed but at different cropping scales. Figure 3.21 depicts the three different scales which are ingested by the proposed 3-CNN ensemble. We appropriately label each of the three networks used to analyze these as S, M, and L (for their corresponding size abbreviations of Small, Medium and Large).

The purpose of the three scales is mainly to train specialized networks for the specific purposes of contextual information at different receptive field sizes. Training a network with any single one of these scales would specialize it in that particular data, but the network would be oblivious to other natural image data with similar structure but not really belonging to a true object of interest, and thus leading it to produce a large number of false positives which would end up affecting the overall detection accuracy. However, the three networks working together as a committee of classifiers produces a much more robust result that is far more resilient against noise, as a true positive hit will require the activation of all three networks simultaneously, simply by integrating contextual information into the system.

Each of the three neural networks produces three output values, which correspond

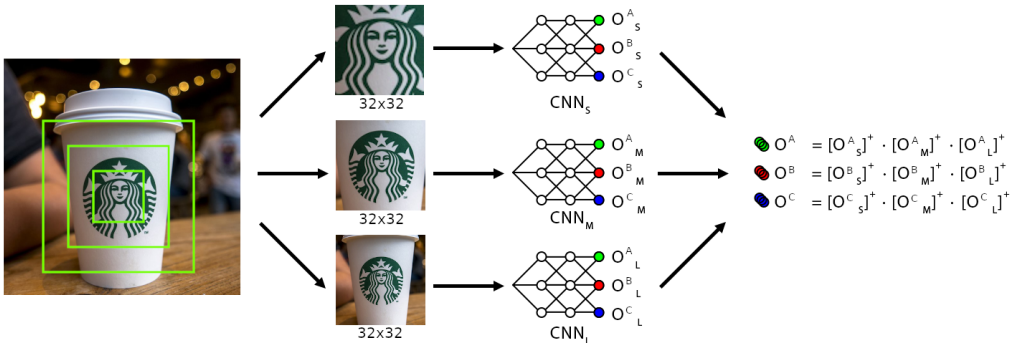


Figure 3.22: Data flow in the inference process of 3-CNN ensemble system, where each network has been trained for three possible classes $K \in A, B, C$.

to the likelihood of each target class having been perceived in that network's input. We denote the output values as O_Z^K, where $Z \in \{S, M, L\}$ represents the network index denoted by its size, and K represents the set of output classes the networks have trained for. Each of these outputs will lie in the $[-1, +1]$ range as the neural networks have been trained with those target values.

To combine the outputs of all three networks as a unified ensemble, we filter each class output with the corresponding values across all three networks, after each one has been linearly rectified. The final outputs of the ensemble are defined by:

$$O^K = \prod_{Z \in S,M,L} \lceil O_Z^K \rceil^+ \tag{3.15}$$

Where $\lceil x \rceil^+ \equiv max(0, x)$, is a linear rectification operation. By passing through only the positive values of each interim output, we avoid interference from multiple negative values, any of which then has the effect of zeroing the final output. Figure 3.22 depicts the process visually.

The net effect of this process, then, is to have all three networks work in tandem, where only the detections for which all three networks are in full agreement will survive. Furthermore, the final output will be weighed by the individual network certainty, and thus regions where all three networks have a high confidence output will outweigh regions where the output distribution is more disparate.

When this 3-CNN system is used in conjunction with the filtering provided by

the belief propagation algorithm presented in Section 3.5, the end result is an extremely robust inference system composed of very light and simple component neural networks.

3.7 Dealing with Scale

State of the art neural networks, like SSD [35] have very interesting multi-scale architectures that integrate pyramidal inference at different image scales. However, such networks are capable of affording to do that due to their heavy and complex backbone architectures. The proposed network architecture is too small to support this kind of operation, so the problem of scale is left as a traditional pyramid of images approach. This essentially means resizing the image at multiple scales and performing inference individually on each of these levels. Multiple overlapping detections from different scales can be resolved with a simple Non Maximum Suppression (NMS) algorithm [50, 51].

3.8 Summary of the Proposed Architecture

This chapter covered multiple aspects of the design of an object detection neural network. A network architecture was proposed that has special connectivity mapping to take advantage of the YUV color space in which ISPs work, as well as the chroma compression that inherently results from this way of representing color information. The color compression in the chroma channels is complemented by contrast normalization performed in the luma channel through a series of efficient convolutional operations.

The proposed system also shows how to extend the basic classification system to a spatially extended inference framework that is capable of performing object detection without requiring specialized layers or additional computational overhead. An algorithm based on Belief Propagation is described that is capable to better analyze the final output shared maps in order to produce cleaner discrete outputs from a series of non-confident classification window results. Finally, another method for filtering false positives was presented, based on a multi-scale ensemble of neural networks, each of which follows the guidelines developed thus far.

The architecture design is inherently designed with computational efficiency in mind. However, the runtime implementation of the software that executes such

a system is just as important. The development of this software will be explored next in Chapter 4.

4 Implementation in Mobile and Embedded Devices

4.1 Application Overview

The application developed for this work consists of the implementation of a neural network for object detection. It is important to note that this is an inference only application, and does not support any training of the neural network. The neural network model is trained offline in a separate system, and the learned parameters are then transferred to the edge application so that it can perform inference on the images it captures from its camera. The image is then pre-processed and fed to the neural network. The results of the network are then computed to retrieve the final detection bounding boxes and this data is sent back to do further processing on these results at the application level. Then the process repeats for the next frame all over again. Figure 4.1 depicts the main loop logic and executing order. As can be seen, this application mainly runs inside of an iterative loop. Therefore, it is important to reduce the time spent on computation as much as possible, which can be achieved by efficient and optimized programming and making better use of the underlying hardware.

4.2 Android Smartphone Platform

As discussed in Section 2.4.2, smartphones are a natural choice for deployment of Edge AI applications due to the overwhelming amount of already deployed devices, but also due to the platform having all of the important requirements to successfully execute an edge computing application.

In traditional computer systems, the various compute components are discrete and separate from one another. A computer's CPU and its RAM memory, for example, are different devices and connect to each other via a memory bus. Similarly,

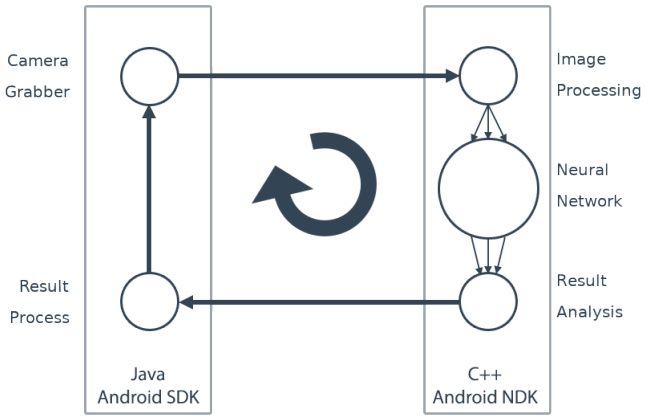


Figure 4.1: The application overview diagram depicting the main loop of the system.

a GPU is usually a separate device that connects with the CPU via a PCIe bus. On smartphones however, microprocessors are developed as a System on a Chip (SoC), a type of integrated circuit that holds all the main processing components required by such a device. In these circuits, the CPU, the ISP, the DRAM memory, and several other blocks are all integrated into a single package. And while specific details might vary greatly from one manufacturer to another, or from one generation to the next, the overall concept remains the same. Figure 4.2 shows the diagram of a common type of SoC.

Of particular interest in these architectures is the Central Processing Unit (CPU). As is common for any type of computer application, the CPU takes on the responsibility of handling main application logic by running instructions compiled into a binary executable. The CPU is good at running sequential instruction sets extremely fast. However, certain types of arithmetic instructions, especially those that can run as parallel computations in a SIMD fashion, can benefit from a component of the ARM Cortex CPU: the *Advanced SIMD* execution units. These components are able to execute vectorized instructions from the NEON Instruction Set. These NEON instructions are capable of executing 4, 8, or 16 parallel vectorized operations simultaneously, thereby achieving a significant throughput speedup when running algorithms that can take advantage of parallel computations of this type. Figure 4.3 shows the memory access configuration when working

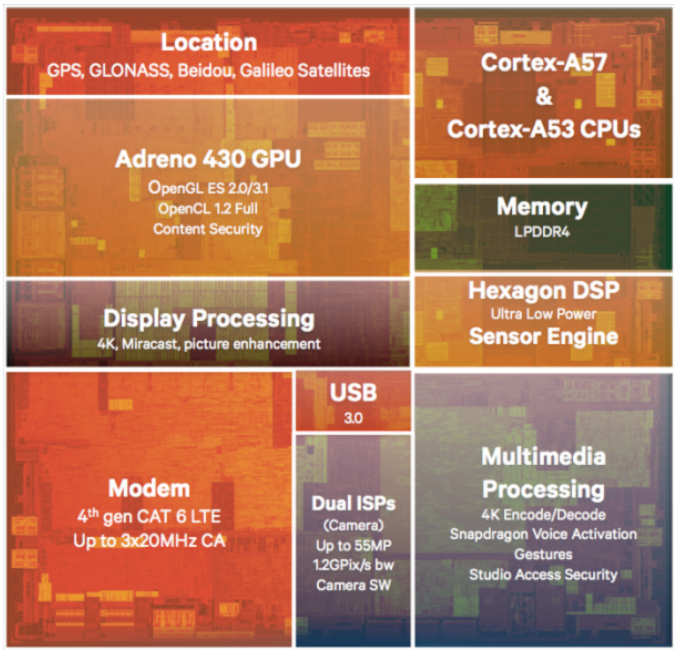


Figure 4.2: The block diagram of the Qualcomm Snapdragon SoC, a very common type of smartphone microprocessor.

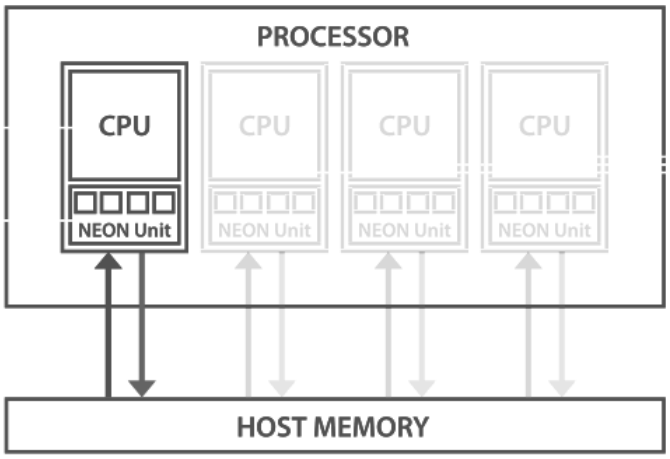


Figure 4.3: The CPU memory access route for mobile SoCs.

with these components. The CPU cores and the NEON units both share access to the same memory bank and as such they can operate interchangeably on the same memory locations.

The other important component in SoCs is the Graphics Processing Unit (GPU). Although the GPU is used primarily to drive on-screen graphics, modern GPUs are excellent parallel processors, capable of running a large amount of concurrent SIMD operations. This is knows as General Purpose GPU (GPGPU) computing, and usually entails using traditionally graphics oriented pipelines in a more generic manner via specialized compute APIs. GPU designs vary largely, but a traditional SoC GPU is usually capable of running up to 32 simultaneous SIMD operations, which would seemingly make it a better choice than CPUs or NEON units. However, GPUs may usually run at different clock speeds, and have different memory access routes than the CPU, even though both share the exact same memory bank. Figure 4.4 visualizes this type of memory routing. GPUs only have access to a specific allocation of the memory bank, and as such, require copying the memory to and from the allocation used by the CPU. This puts an additional overhead on the execution of GPGPU code as memory bandwidth in these SoCs is limited, and must be managed carefully. Therefore, GPGPU code is usually favorable only when large amounts of operations can execute continuously on a single allocation of data, such as to reduce the number of memory copies that must be performed between the CPU and GPU address spaces.

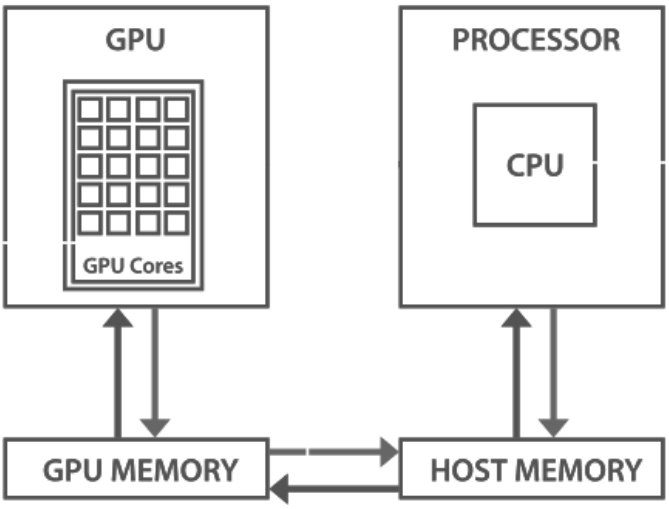


Figure 4.4: The GPU memory access route for mobile SoCs.

The Android ecosystem provides an easily accessible Software Development Kit (SDK) that allows easy and often guided development of user applications through the Java language. The Android SDK therefore gives the developer access to virtually all functionality that an application might be required to use. However, the Android Java SDK sits on top of native C++ libraries and the Android Runtime, and as such does not have low level access to many hardware level instructions.

The NEON instruction set, specifically, requires development at the native C++ level, as there exist no bindings to access these instructions from the Java SDK layer. To handle such kinds of programs, the Android platform gives developers access to the Native Development Kit (NDK). This provides a way of executing low level hardware instructions via C++ code, which can be compiled into a library, which can then be accessed from the Java SDK layer.

Similarly, to access the GPU, it is often necessary to make use of GPGPU SDKs that are usually accessible only through native layer. This is the case of OpenCL, a parallel programming language designed to execute SIMD kernels in parallel compute devices such as GPUs. The OpenCL kernel is compiled at runtime by an OpenCL driver specific for the SoC GPU being targeted.

Yet another possible compute unit is the Digital Signal Processor (DSP), traditionally aimed to processing audio signals, some SoCs expose general compute

capabilities that can be programmed and executed on these units, by providing a low level C++ SDK. Such applications are usually tied to a specific DSP from one particular SoC manufacturer, and as such are hardly portable to other types of smartphone devices.

Therefore, to have access to all these low level routines, it is necessary to generate the application via the Android NDK in order to make the best possible use of the hardware resources available in the smartphone's SoC. However, an inevitable problem of this approach comes about by device fragmentation. As explained in Section 2.4.2, device fragmentation occurs when there is an extremely large variety of devices. As a result of this, the development efforts must be largely multiplied as it is necessary to support a wide array of different hardware configurations, many of which might not even support NEON or OpenCL instructions to begin with.

To tackle this problem, the Android platform has introduced an abstract parallel programming language known as Renderscript. As the name suggests, this started as a graphics programming language, however it evolved into having GPGPU support, and supports generic non-graphical workloads. The Renderscript language abstracts many of the problems related to device fragmentation by providing a unified language framework which can execute on any of the available compute blocks of the smartphone it is running on. The development effort is shifted towards the manufacturer to develop Renderscript-specific drivers that support all or most of these compute units. Possible compute backends for Renderscript applications include GPU, DSP, and NEON, with a standard generic fallback of executing the program as sequential instructions on the CPU when no other hardware specific drivers are available for the device it runs on.

It is interesting to note, however, that Renderscript does not allow targeting a specific compute unit. Rather, the Renderscript drivers choose at runtime the most suitable compute backend to run on, depending on the type of the kernel, the type of memory it needs to operate on, and the current load on the device.

As a result, Renderscript was chosen as the language of choice to develop and experiment on.

4.3 Movidius VPU Platform

The Intel Movidius VPU platform consists of an embedded device designed specifically to accelerate computer vision and neural network inference tasks. The device

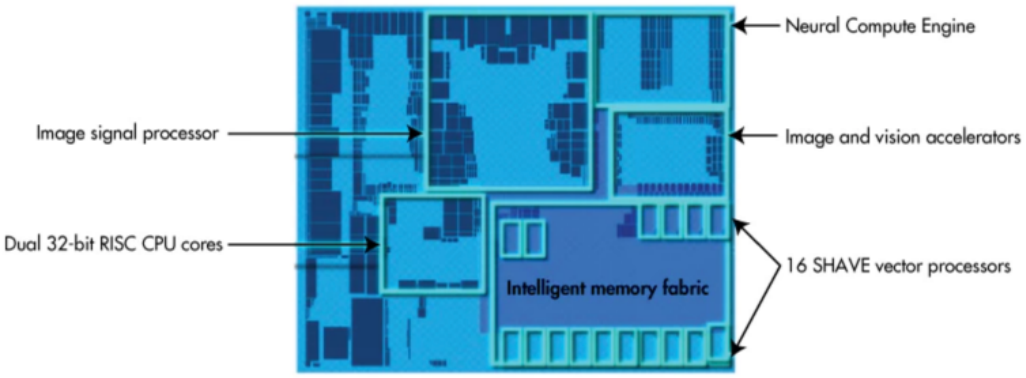


Figure 4.5: The block diagram of the Intel Movidius Myriad X VPU, a computer vision accelerator embedded device.

is capable of approximately 1 TFLOPS of computing capability, but only consumes 1 Watt of power. This high efficiency makes it an excellent deployment environment for Edge AI implementations.

Figure 4.5 shows the hardware block diagram of a Myriad X VPU, the current generation device of this platform. Primarily, the VPU consists of a dual core CPU capable of performing general computation tasks. The VPU also has several other co-processor cores, such as two Neural Compute Engine (NCE) cores, which are MAC operation accelerators capable of performing the most fundamental arithmetic operations of neural networks extremely efficiently. The platform also has 16 SHAVE SIMD processors for general vectorized computation, extremely useful to develop custom image processing algorithms. The VPU also has an integrated ISP to process the image stream from a camera.

The Myriad X VPU platform is available on different deployment packages depending on the use case. For developers, the most accessible form factor is as a USB device known as the Neural Compute Stick 2 (NCS2). This device connects to a standard computer via the USB connector and acts as a neural co-processor to offload the main system from any operations that can run on it. Another choice is the MV0235 developer kit system, a standalone board that doesn't need an accompanying system to operate. Furthermore, this developer kit board has an embedded camera which makes it very suitable to develop and test computer vision tasks with. Besides developer packages, other production ready form factors also exist, depending on the application.

The VPU does not run an operating system. Everything, including the custom application developed for it, runs via programmable firmware that must be flashed on the board. This firmware is developed via an SDK known as the Myriad Developer Kit (MDK). Traditional C++ code can be written with it which will run on the VPU's CPU. To access the SHAVE or ISP cores, specialized APIs within the MDK must be used, with some operations are available only through assembly instructions. The NCE cores are only programmable via assembly level instructions. However, OpenVino provides a framework to perform inference of neural networks on the Myriad X, which makes use of these NCE cores.

4.4 Optimized Parallel Programming

To make use of Renderscript as a method to implement the object detection neural network, it is necessary to develop multiple kernels to support each of the possible network layers. This is mainly the convolutional layer, the max-pooling layer, and the linear layer. However, an efficient and optimized implementation introduces some additional requirements, which will be explored in more detail in this section.

Several algorithm optimizations are performed to these kernels in order to obtain the most optimal implementation of the layers. The optimizations are built into various stages of the kernel logic, but will be explored here individually. These optimizations have two main objectives: (i) reduce the number of instructions that need to be executed to carry out an algorithm, or use instructions that can be completed in fewer cycles, and (ii) increase the arithmetic intensity of an algorithm in order to reduce the memory bandwidth required to transfer data between hardware units.

4.4.1 Loop Unrolling

Image processing operations are highly iterative in nature. For example, a naive implementation of a convolutional neural network layer can involve up to 6 levels of nested loops. With the help of parallel SIMD programming, a lot of these loops can be removed, as they all act independently from one location of the image to another. However, the inner two nested loops are not as easy to parallelize as they act on a single pixel of the image data. Listing 4.1 shows two equivalent pieces of code, one inside a for loop, and one unrolled as individual statements.

Although syntactically they perform the same operations, the second piece will usually execute faster as it does not have the overhead of loop control.

Listing 4.1: Example of unrolling a for loop over three iterations.

```
void loop(int loops, float * a, float * b, float * c) {
    for (int i = 0; i < loops; i++) {
        c[i] = a[i] * b[i];
    }
}

void unrolled_loop_3(float * a, float * b, float * c) {
    c[0] = a[0] * b[0];
    c[1] = a[1] * b[1];
    c[2] = a[2] * b[2];
}

loop(3, a, b, c, d);
unrolled_loop_3(a, b, c, d);
```

4.4.2 Kernel Specialization

In cases where the number of iterations is constant, the compiler will often do loop unrolling automatically when the application is built. However, most network layer implementations are either data dependent or configuration dependent. This means the number of iterations may not be known at the time the application is built. For example, a generic convolutional layer kernel may have its inner loop loop over 9 or 25 iterations, depending if the layer is using a 3×3 or 5×5 convolution kernel.

In such cases, loop unrolling and other configuration specific optimization can still be performed simply by specializing the kernel and having multiple implementations, one for each possible configuration. This reduces the flexibility of a kernel, as its execution will no longer be dynamic and adaptable based on configuration parameters. However, this will allow each possible kernel configuration to execute in the most optimal manner possible.

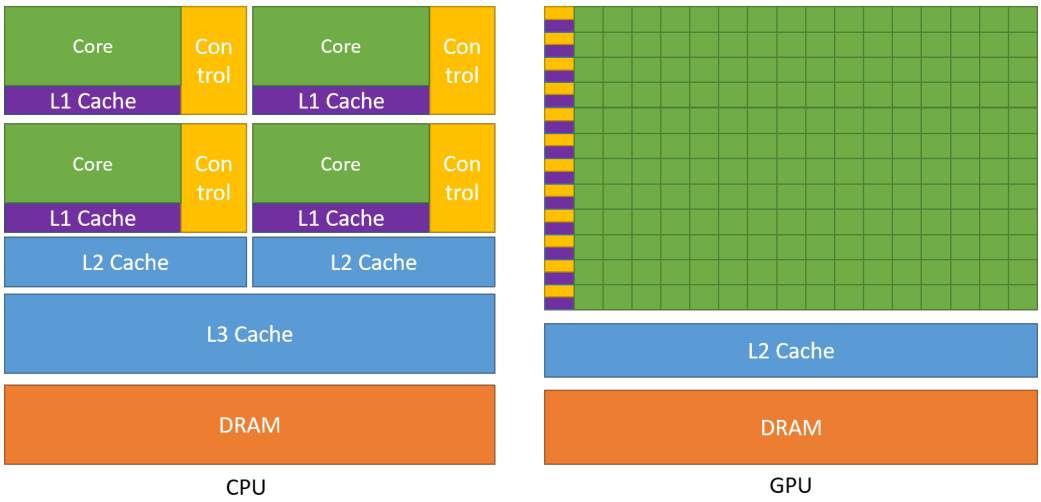


Figure 4.6: The usual configurations of the different memory levels available to either CPU or GPU cores.

4.4.3 Memory Access Patterns

The hardware compute units in a mobile SoC have access to different memory levels. There is the global DRAM memory bank which is accessible to both CPU and GPU units, and is usually implemented as DDR memory. This memory space usually ranges between several hundreds of megabytes to a few gigabytes. Bandwidth and latency to access this memory bank can often be a bottleneck, as DRAM requests can usually take a few hundred cycles to operate. Although it is possible to do latency hiding by more appropriately scheduling and interleaving memory requests and arithmetic operations, at some point the overhead of accessing this memory bank too frequently becomes inevitable.

Closer to the compute unit there usually are one or more levels of memory cache. This memory area is much faster to access, with considerably lower latency and higher bandwidth to the compute unit. However, the size of these caches is very limited, and usually ranges between several hundreds of kilobytes to a few megabytes. Figure 4.6 shows the different levels of memory that may be present in an SoC for different compute unit types.

Memory caching is handled by the hardware memory controller, and as the name suggests, it is used to keep a temporary copy of any data requested from the main

global memory bank. This is especially useful due to the nature of DRAM memory requests, which only work in chunks of 128 bytes. That is to say, requesting any element in DRAM will always return 128 contiguous bytes. Keeping this chunk of memory in cache can help to speed up access to other contiguous memory elements, as a single request to DRAM will bring all 128 bytes to the faster cache unit, instead of requiring a large number of individual DRAM requests. This, of course, requires algorithms that are better optimized to make good use of this characteristic. Especially since the size of cache memory is limited, and if the cached data is not used quickly, it will be overwritten by other DRAM requests.

This effect can be most easily appreciated by adapting the order in which memory is accessed, especially when dealing with images or general matrices are stored in DRAM. There exist two methods to store 2D matrices in memory: (i) row major format, or (ii) column major format. Figure 4.7 shows the difference between both. It is customary to store image data in the row major format. The underlying reason is related to the per-line readout mode of CMOS image sensors, the raw data collected from the sensor is dumped line by line to the target memory allocation, and from there on, all following operations in the ISP happen in this same row major data format. As a result, the image as obtained from the camera grabber, regardless of the pixel packing format, will always follow row major format as well.

It therefore follows that using a memory access pattern where the data requested from DRAM for subsequent operations is in the same row as the current operation will greatly speed up the operation of the algorithm, as the total number of DRAM requests will be lower.

4.4.4 Layer Fusion

An additional level of memory access consists of the registers. Registers are located within the actual core unit, and as such, they are the closest and therefore faster memory type. However, they are also extremely limited, as depending on the type of core, these register banks may be only a few bytes in size, that is, there may only be a very small number of registers, as few as 16, available to each compute unit. Conceptually, these registers hold the values of variables that individual instructions are operating on. Some registers, such as the Program Counter (PC) are restricted and the hardware has a specific use for them, but others are available and normally assigned depending on the instruction set generated by the compiler.

As this type of memory is the fastest and has practically zero latency, it is essential

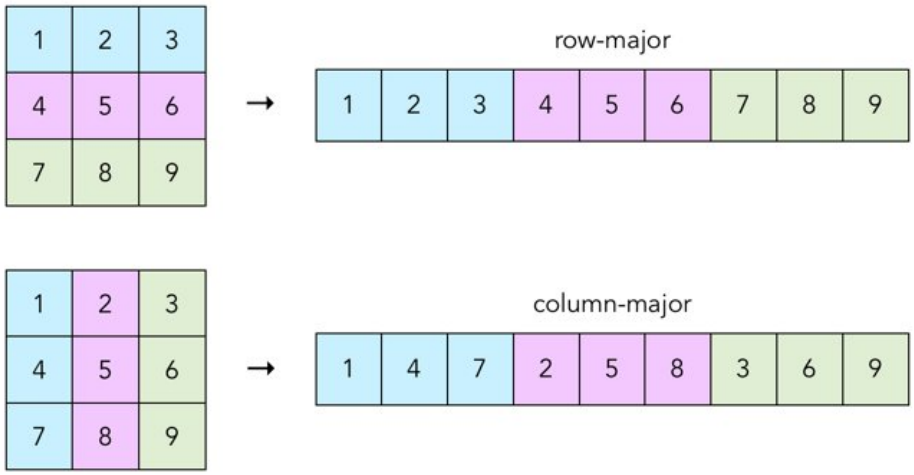


Figure 4.7: There are two possible ways to store matrix information in memory, depending on the order of indexing.

to make as much use of it as possible. It is completely up to the compiler to have the final decision on how registers are used and when the values of a register are reused for the next operation, or they get overwritten by a different variable requested from cache or DRAM. However, some decision in the code design may help to maintain memory active in registers for longer, which can have a visible impact on the overall throughput of the algorithm. This technique is known as register reuse.

Essentially, there are two ways to achieve register reuse. First, it is good practice to limit the number of local variables in a function. By reducing register pressure, it is far more likely that the values being operated on will be kept in local core registers rather than relying on accessing cache or DRAM.

Second, it is essential to try to keep intermediate algorithm results in local variables for as long as possible, instead of writing them back to arrays or pointers. This will guide the compiler to making more efficient register assignment choices. Of course for trivial operations, underlying compiler optimizations may already do this, but one instance where this practice is most apparent is in layer fusion.

Fusing operations essentially means doing multiple operations at once. Listing 4.2 shows a simple example of how a fused multiply and accumulate operation can replace individual multiply and accumulate functions. Due to the 256 array size it operates on, each non-fused function would be forced to request multiple cache lines to fulfill the operation. As the *mul_32()* and *acc_32()* operations are executed sequentially, it will force access to DRAM to write the results of the former and then again to retrieve the inputs of the latter. In the fused *mac_32()* function, however, values are read only once and written only once. Thus reducing the number of DRAM requests.

Listing 4.2: Simplified example of fusing a MAC operation in order to encourage register reuse and maintain intermediate values locally, thus reducing the frequency of memory access to cache or DRAM.

```
1  void mul_32(float * a, float * b, float * c) {
2      for (int i = 0; i < 32; i++) {
3          c[i] = a[i] * b[i];
4      }
5  }
6
7  void acc_32(float * a, float * b, float * c) {
8      for (int i = 0; i < 32; i++) {
9          c[i] = a[i] + b[i];
10     }
11 }
12
13 void mac_32(float * a, float * b, float * c, float * d) {
14     for (int i = 0; i < 32; i++) {
15         d[i] = a[i] * b[i] + c[i];
16     }
17 }
18
19 mul_32(a, b, res);
20 acc_32(res, c, d));
21
22 mac_32(a, b, c, d);
```

This concept is even more important in the fusion of neural network layers, as

the operations are obviously more complex than simple MACs, and memory access patterns can be quite more complex. A good use case of layer fusion is the convolutional layers. In the proposed neural network architecture they are always followed by max-pooling layers. Compared to convolution, the max-pooling operation is much simpler and has lower arithmetic intensity. Therefore, requiring a full DRAM request cycle in between the convolutional and max-pooling layers can be quite inefficient. Fusing these two layers, therefore, yields significant throughput benefits as it will essentially cut in half the amount of DRAM requests.

4.4.5 Efficient Configuration Data

As can be seen, optimizing memory access is extremely important for optimization. In the case of neural network layer implementation, this is not limited only to the input and output image data, but also to the parameter data that must be operated with. As part of this work, a binary data format was developed that optimizes the access to the layer configuration and learned parameters at runtime. Figure 4.8 depicts the structure of this data format.

This data format is first used to store in a file the network configuration and parameters retrieved from a neural network training procedure, at which point the file can then transferred to the mobile device. During launch of the application, the file is read and its contents are mapped directly to memory, so that the structure of the data in DRAM follows the exact same structure.

The beginning of the data structure holds a header section containing the network and layer configuration. This can be considered metadata that holds the information required at program start to configure and layout the Renderscript kernels that will need to be executed, and the order in which they must operate.

The main part of the data structure holds the payload section containing all the learned parameters of the network. In the case of convolutional and linear layers, these are the weights for each neuron. The order of weight values is arranged in such a manner that allows them to be read in a row-major format. The weights for different neurons are also positioned in the same order as the execution in the corresponding Renderscript kernel is performed. This allows for better matching and aligning of the weight values with the corresponding image elements they must operate on, while minimizing the number of DRAM requests that are necessary to fetch weight data.

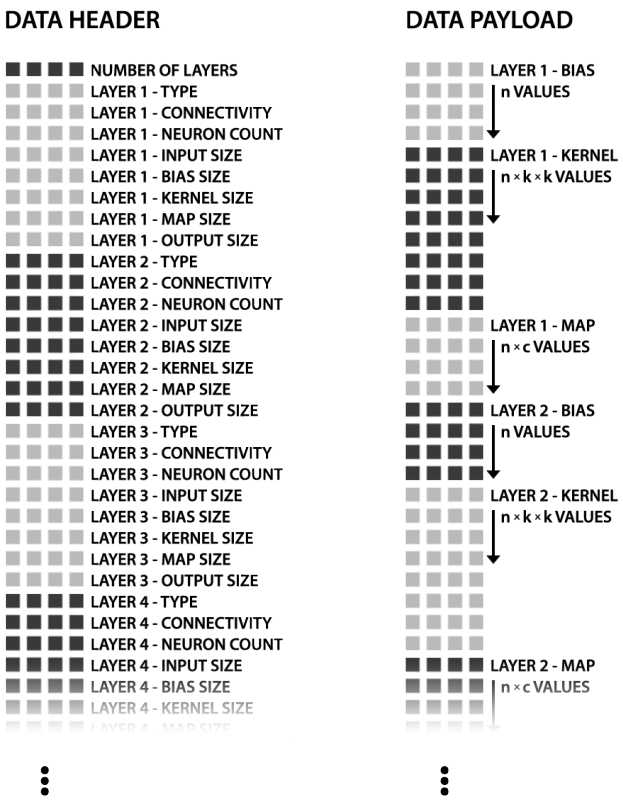


Figure 4.8: Network configuration and layer parameters are stored in a binary file that can be consumed by the mobile application.

5 Experiments and Results

The proposed neural network architecture described in Chapter 3 and its implementation as reviewed in Chapter 4 are explored in an experimental manner in this chapter, to determine its validity and applicability to an object detection task targeting low energy devices.

Two types of experiments are carried out: (i) tests on the accuracy of the neural network, to verify how different architecture design decisions impact the final accuracy of the system, and (ii) tests on the runtime of the neural network, to verify how different architecture and implementation designs affect the efficiency of the system.

To carry out accuracy experiments, a baseline neural network is trained which is used to validate various metrics related to the capacity of the system to properly detect objects. For this, a simple object detection task has been devised, where the purpose of the neural network is to find and recognize objects among a number of possible classes in sample images.

To perform the runtime experiments, several mobile and embedded devices are used to benchmark the neural networks deployed on them. These devices provide a broad range of compute capabilities and represent common edge devices which could be used to deploy this kind of system. The neural network is deployed with the runtime optimizations previously outlined.

5.1 Object Detection Baseline Model

This section explores how a baseline model is created to run the various experiments under. As a toy example, a company logo recognition system is devised. This is created with purely synthetic data which results in an easy to produce training dataset.

The idea of using company logos was of particular interest as it inherently requires recognizing relatively complicated forms and shapes. However, it is easy to see how this seemingly trivial process can just as well extend to many other interesting applications, such as recognizing traffic signs for advanced driver assistance applications, or recognizing warning and content labels in automated package sorting applications.

5.1.1 Synthetic Data Generation

Convolutional neural networks, as with just about any other machine learning algorithm, rely greatly on both the quantity and quality of the data used for their training. The accuracy of a CNN is directly related to its ability to extract the correct representation features from input images, as well as the ability to correctly generalize to never before seen inputs. The model can only acquire these abilities through a vast amount of training data that thoroughly covers the expected input space that the model will be exposed to at run time.

In this sense, input space refers to the variance observed in the image of an object under different photographic conditions. It is important to note that this variance is not trivial at all. Under varying conditions, the numerical values that make up the image pixels not only increase or decrease in response to the brightness perceived, but instead can go through much more complex changes, such as hue shifts due to white balance issues arising from different light sources, or color saturation differences due to varying camera sensitivity conditions.

This obviously only takes into account different illumination effects on the object. More importantly, however, due to viewpoints differences formed by the relative position between the target object and the camera, both in distance and in the angle between them, the object will cast a very different 2D projection of itself onto the camera's view at every state. As the final model should be ideally capable of recognizing an object regardless of the viewpoint, it is quite important to train the system to recognize it under different perspective views. Additionally, there is the issue of external clutter, as sample images will rarely portray the object isolated but rather will have other clutter in the background that the object detection system should similarly learn to ignore.

When facing an image recognition task, one of the first steps that is must be taken is the gathering a large representative dataset for training the model and testing its

performance. This should preferably be done under differently unique conditions as described above, and carefully labeling each sample image in accordance to the target image found in it. Additionally, when dealing with naturally photographed images, each sample object needs to be bounded to an area matching as close as possible the area where the object itself is located. The data collection step, therefore, presents a laborious task in any image recognition project, especially if the data is not available beforehand. This is usually one of the most time demanding procedures in building such as system, as natural images of target objects are difficult to acquire, and even more so if it is required to systematically capture all possible variations that the object can be seen under.

The recognition task developed for this benchmark, natural images of company logos, presents a particularly difficult problem – in that such images are not readily available, at least not in the volume required for fully training a neural network. This makes it highly impractical to manually photograph the vast amount of uniquely different and accurately labeled logo samples for an ever increasing amount of target training images.

For this reason, a method is proposed to artificially generate large amounts of samples through a computer program that can output a virtually limitless amount of such images. Starting with a clean representation of each logo image to be trained, several thousand samples can be created from this seed image to be used as training data. By performing random perturbations in each of the possible distortions described, the final samples will contain all sorts of small and subtle variations that will make them distinct from one another, with the full set of them covering an ample region of the expected input space.

Perspective Projection of Planar Objects

Company logos are a perfect example of a planar object, which can be represented as a 2D image freely rotated in 3D space. The image shape, regardless of the variable viewpoint of the camera, will maintain planar invariance within itself. As a result, viewpoint variance can be simulated through a series of linear transformations applied to the seed image. As it is necessary to simulate a pinhole camera view, the full perspective projection transformation [52] must be used instead. This can be seen in Figure 5.1.

A perspective projection matrix based on the pinpoint camera model with the Viewpoint positioned at the origin $\vec{O}$ is applied to the seed image whose position

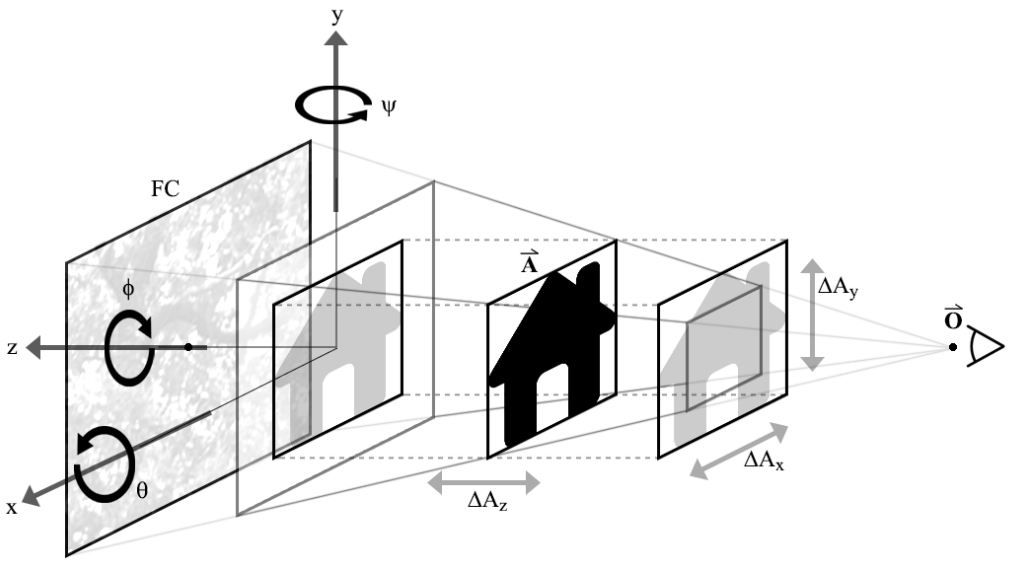


Figure 5.1: The perspective transformations performed in order to simulate different orientations of the object over the background.

is denoted by the vector $\vec{A}$, whose components A_x, A_y, A_z denote the values for the translation in the x, y and z axes, and the rotations in each of these axes is given by θ, γ, and ψ respectively. These six values are randomized for each new data sample generated. The resulting $\vec{B}$ representation is the standard perspective projection matrix applied to the seed image, as given by:

$$\vec{B} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & cos(\theta) & -sin(\theta) \\ 0 & sin(\theta) & cos(\theta) \end{bmatrix} \begin{bmatrix} cos(\gamma) & 0 & sin(\gamma) \\ 0 & 1 & 0 \\ -sin(\gamma) & 0 & cos(\gamma) \end{bmatrix} \begin{bmatrix} cos(\psi) & -sin(\psi) & 0 \\ sin(\psi) & cos(\psi) & 0 \\ 0 & 0 & 1 \end{bmatrix} \vec{A} \tag{5.1}$$

Each of the six variable values are limited to a pre-defined range so as to yield plausible viewpoint variations which allow for correct visual recognition. The exact ranges used will vary on the implementation requirements of the application, but in general, the z-translation limits will be approximately ±30% of the distance between the seed image and the viewpoint, the x and y translations will be ±15% of the width of the seed image, and the θ, γ and ψ rotations will be ±30° around their corresponding axes. The Z Volume depicts in particular the effect of trans-

lation along the z axis (the camera view axis), where the seed image can be seen projected along the Viewing Frustum at both the Near Limit and Far Limit of the z-translation parameter.

Clutter and background simulation is performed at the Far Clipping Plane of the projection, a different texture is placed at this plane for each of the generated sample images. This texture is selected randomly from a large graphical repository of random natural images, such as randomly sampled patches from images in the COCO dataset [41]. The purpose of this texture is to create synthetic background noise and plausible surrounding context for the target shape, where the randomness of the selected texture allows the neural network to learn to distinguish between the actual traits of the target shape and what is merely clutter noise surrounding the object.

Illumination Variance

To account for the many possible variations in lighting, the image samples should also have their color information shifted in several different ways.

Initially, the image is adjusted so that the black and white point values are shifted from their original levels. These changes have the effect of simulating different camera exposure levels, which are affected not only by the actual amount of light shining on an object and then reaching the camera sensor, but also by the automatic exposure adjustments performed by the camera's internal hardware in response to the global light levels received.

Next, the gamma factor of the image is adjusted randomly, either increasing or decreasing it, such as to simulate the effects of different sensitivity levels of the perceived image by the camera's sensor.

Image hue also needs to be adjusted so as to simulate the difference between the color temperature of a light source and the white balance setting under which the device's camera may be operating. Usually a subtle change not exceeding 10°in either direction is sufficient.

Finally, color saturation can be reduced by a random amount, as different exposure and lighting combinations can lead to desaturated colors in the images captured by a camera.

This completes the data generation procedure. Figure 5.2 shows a sample of what the final training data set looks like on nine different logo classes, and one

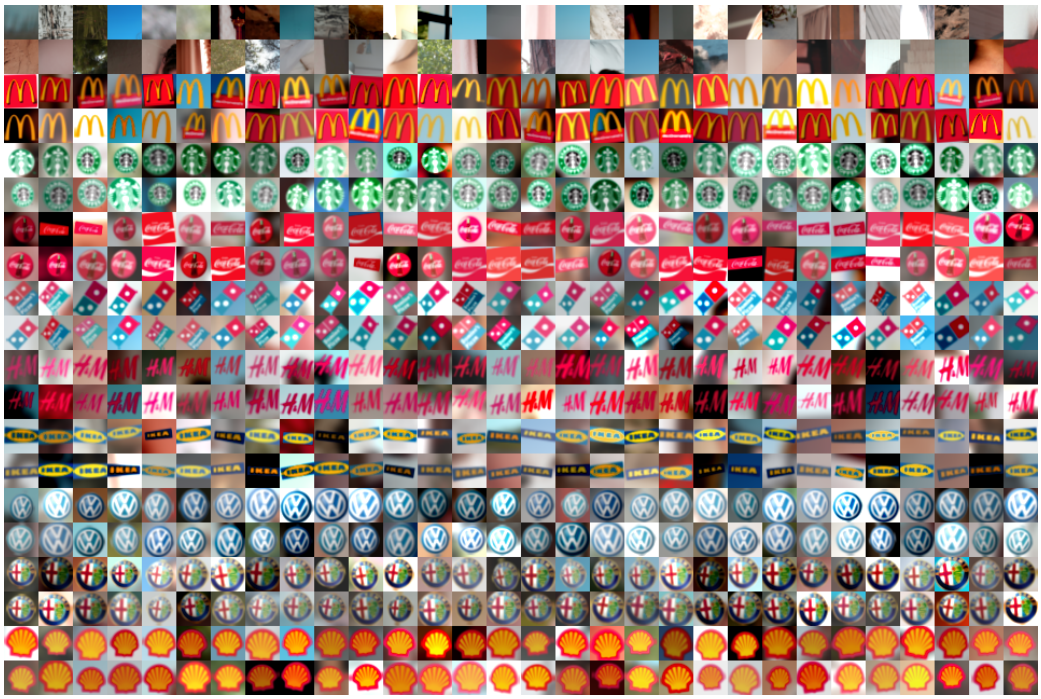


Figure 5.2: Example of the final set of artificially generated data samples with all of the randomly assigned distortions applied.

background-only class. Note that some classes use multiple seed images to simulate the possible design differences that a logo can take.

Given that the proposed network architecture is essentially a classification network, the training dataset is created as square patches with the target object covering most of the area in the generated image. The test dataset is generated in a similar manner, but with a larger random background surrounding the target object. The exact location of the object within this extended image is recorded as an annotation to the image. This information will be used to measure the localization performance of the network when performing object detection through shared maps.

5.1.2 Network Architecture

Once the training dataset is ready, the next step is to define a configuration for the neural network. The input image size is the first, and possibly most important

Table 5.1: Multiple input size configuration values and its effect on the CNN classification performance.

	Parameters			**Performance**	
Input Size	Kernel Size	Pooling Size	No. of Operations	SoC Compute Time (ms)	Mean Error (%)
16×16	3×3	2	110,954	0.02	12.54
24×24	5×5	2	481,418	0.08	1.32
32×32	5×5	2	1,121,418	0.17	0.44
48×48	7×7	2	4,907,882	0.49	0.15
64×64	7×7	4	10,038,378	1.77	0.18

parameter that must be set in the CNN architecture. It defines the data size over which the network will run, requiring all tested images to first be resized to match these dimensions.

A fine balance must be found when selecting the input data size, the reason being that this setting will affect both the performance and the accuracy of the model. If the size is too small, the model will be fast, but it won't have enough pixel data to make correct decisions on what exactly it is viewing, so classification accuracy will drop. if the size is too big, there will be a lot of data to work with which will aid the model's accuracy rating but more calculations will be required and performance may be too slow for real time usage.

Table 5.1 shows a few different values for possible image input sizes. The kernel size of the convolutional layers as well as the window size for the max-pooling layers are varied for each tested size. The number of arithmetic operations required to execute the full CNN with these parameters is calculated. As can be expected, the number of calculations is proportional to the time it takes to process the neural network. The classification accuracy on the test dataset can be seen in the last column. This provides sufficient evidence that a 32×32 input image size is the most optimal for this particular system.

5.1.3 Training

The network is trained in such a way that it will learn to distinguish target objects from background noise – something essential for a system that will be constantly

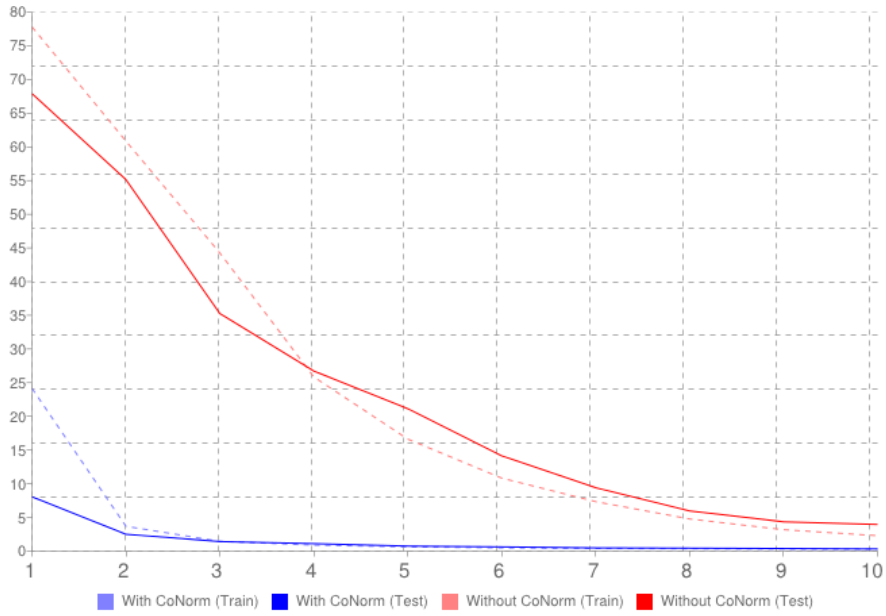


Figure 5.3: The effect of contrast normalization on the mean error learning curves.

exposed to non-activating visual stimuli. The system should therefore, not only learn to distinguish one trained logo from another, but also whether the image it currently sees is a logo at all. This is an essential requirement when extending the classifier to performed shared map object detection.

The training targets, then, are ideal one-vs-all vectors, where the i-th element of the vector corresponding to the class index takes on a value of $+1$, while all other elements are assigned a -1 value instead. The (-1,+1) interval, being zero-centered, helps in the training phase by having evenly and symmetrically distributed target values.

The network is trained through Stochastic Gradient Descent [53], where the configuration parameters are updated after every single training image. This requires the image data set to previously shuffled so as to avoid any biasing that may occur from training over the same class for too many sequential steps. Figure 5.3 shows the mean classification error over 10 training epochs of two learning runs. This figure also shows the effect of using contrast normalization on the training procedure. Not only is the final accuracy better, as will be discussed in more detail in the experiments below, but the training optimizer converges quicker as well due to the better distribution of data values.

Table 5.2: Confusion matrix and global mean error of the DCNN, using the proposed classification backbone network.

	Classes										**Error**
	BG	C1	C2	C3	C4	C5	C6	C7	C8	C9	
BG	-	14	6	16	0	8	15	20	0	3	2.562%
C1	6	-	0	0	0	0	0	0	0	0	0.969%
C2	2	0	-	2	0	0	0	0	0	0	0.125%
C3	3	0	0	-	0	0	0	19	0	0	0.687%
C4	0	0	0	0	-	0	0	0	0	0	0.000%
C5	4	0	0	0	0	-	0	0	0	0	0.125%
C6	8	0	0	0	0	0	-	0	0	0	0.250%
C7	21	0	0	6	0	0	0	-	0	0	0.844%
C8	0	0	0	0	0	0	0	0	-	0	0.000%
C9	0	0	0	0	0	0	0	0	0	-	0.000%
									Classification Error		0.478 %

Table 5.2 gives a summary of the baseline accuracy and confusion matrix achieved with this approach.

5.2 Results

The test devices used to report runtime results are:

- **SoC:** OnePlus 7T Smartphone with Qualcomm Snapdragon 855+; an SoC with GPU capable of 110 GFLOPS
- **VPU:** Intel Movidius Development Kit MV0235 with a Myriad X; a VPU capable of approximately 1 TFLOPS

The SoC test application is developed for the Android OS using a Renderscript compute backend. The VPU test application is developed with the Myriad MDK framwork and OpenVino.

Unless otherwise noted, the baseline architecture configuration is the described in Figure 3.13, with a layer configuration of:

$$12C5 + 12MP2 + 32C5 + 32MP2 + 128L + 2L$$

There is no defined criteria by which this configuration was chosen. As can be expected, the larger a network is, the better it will perform, but also the slowest it will execute as. So in practice it's a matter of setting a compute or energy budget under which the system needs to run, and selecting the configuration that best fits within that constraint.

The selected architecture uses bottleneck YUV channel mapping for the first convolutional layer, and 1/3 random connectivity mapping between the first and second convolutional stage. The shared map configuration uses a T_0 window stride of 8×8. The input image size of the network is 32×32 but due to the subsampling performed by contrast normalization, the effective receptive field input size is 64×64. Belief Propagation and a 3-CNN ensemble system is used by default, with each network in the ensemble having the configuration described so far.

The object detection experiments are performed on a test dataset fold consisting of square images of size 640×640. Each image has randomly a number between 1 and 5 logo objects, at random positions and scales, all generated synthetically on top of background images following the same procedure described in the image above.

Inference is performed with a 4 scale image pyramid to handle objects of different sizes. The pyramid levels are created at factors of one half starting from the original size: 320×320, 160×160, and 80×80.

No threshold is set for any of the experiments to better analyze the performance of the models at different operating points. Therefore, mean Average Precision (mAP) [54] is used as metric to report accuracy of the object detection tasks. The mAP metric is calculated with an IOU of 0.5 by default in all cases.

In all of the following experiments, several variations are tested to determine best configurations, algorithms, or any other design decisions in the object detection algorithm. The default reference configuration is highlighted in in bold in the tables of all results.

Most of the results reported follow a similar structure: for each configuration tested, an mAP accuracy metric is provided and execution run times in both reference platforms. An average over 100 test runs for each of these configurations was taken as the execution time for each of the methods and platforms reported.

Table 5.3: Results of using different pre-processing methods for the input image.

Algorithm	**mAP**	**Execution Time (ms)**	
		SoC	VPU
None	67.8	—	—
Feature Scaling	70.2	0.3	0.6*
Mean Subtract	69.3	0.3	0.6*
Histogram Equalization	85.0	3.7	1.6
Min/Max Normalization	75.2	3.4	1.9
Contrast Normalization	**87.4**	**2.2**	**0.8**

5.2.1 Image Pre-Processing Algorithm

The pre-processing algorithm proposed in this work is Contrast Normalization. However, the suitability of this algorithm is tested against other alternative methods for image preparation, or a lack thereof. Table 5.3 shows these results.

As can be seen contrast normalization clearly results in a higher mAP accuracy metric, while maintaining reasonable compute times in comparison to the other options. It's interesting to see that the Myriad X VPU has a much more efficient implementation of separable convolutions, so the effect of contrast normalization in that platform is extremely small.

Histogram normalization is a close rival, but the runtime of this algorithms is considerably higher, due to global statistics needing to be collected for the entire image, which is an expensive operation in parallel processing. A similar effect can be seen on min/max normalization as this requires a global reduction operation to be performed which is similarly expensive.

The execution time reported is for the pre-processing algorithm by itself, without taking into account the neural network, as the object detection system is identical for all configurations presented here. Note that the Feature Scaling and Mean Subtract algorithms in the Myriad X VPU were implemented via OpenVino operations which adds a fixed overhead, so the reported times are not representative of the operation when fused to the rest of the neural network.

Table 5.4: Results of using different color spaces for the input image.

Color Space	Contrast Normalization	mAP	**Execution Time (ms)**	
			SoC	VPU
RGB	None	67.8	—	—
RGB	3 Channels	82.9	5.8	1.9
HSL*	None	69.3	0.6	0.4
HSL*	L Channel	83.2	2.7	0.9
HSL*	3 Channels	79.7	6.1	2.1
LAB*	None	71.3	1.2	0.7
LAB*	L Channel	85.2	3.1	1.2
LAB*	3 Channels	83.3	6.7	2.5
YUV	None	71.0	—	—
YUV	**Y Channel**	**87.4**	**2.2**	**0.8**
YUV	3 Channels	83.1	5.8	1.9

5.2.2 Image Color Space

Table 5.4 shows the effect of using different color spaces. For each color space tested, contrast normalization was applied on a different subset of color channels, or on none at all, as the effects of color space are intertwined with the choice and application of pre-processing.

Although these experiments are done on synthetically generated data, which does not pass through the camera ISP, the effects of color space conversion are not reflected in the case of the YUV experiments, as this color space would normally be readily available from the ISP.

Note that the HSL and LAB color spaces do include conversion overhead in the execution time however, due to color space conversion, as these are not supported by the standard imaging pipelines. Execution times therefore include both contrast normalization when applicable and color space conversion when necessary.

As can be seen, the YUV color space with contrast normalization applied only on the Y channel resulted in superior mAP values. Applying contrast normalization on all three YUV channels not only is more computationally expensive, but has a detrimental effect on the accuracy results.

Table 5.5: Results of different bottleneck connectivity mapping between the input image and the first convolutional feature extractor.

Color Space	Channel Connectivity	mAP	**Execution Time (ms)**	
			SoC	VPU
RGB	R:4 G:4 B:4	82.9	17.4	1.9
RGB	R:2 G:8 B:2	82.7	17.4	1.9
RGB	R:8 G:2 B:2	81.3	17.3	1.9
RGB	R:12 G:12 B:12	87.8	28.7	3.3
YUV	Y:4 U:4 V:4	82.8	17.3	1.9
YUV	**Y:8 U:2 V:2**	**87.4**	**17.4**	**1.9**
YUV	Y:10 U:1 V:1	86.7	17.4	1.9
YUV	Y:12 U:12 V:12	88.1	28.7	3.4

When no contrast normalization is applied, the YUV color space still has an advantage on RGB, but this is an effect of the first stage Y channel connectivity, as will be further explored in the next experiment.

5.2.3 First Stage Connectivity Mapping

Experiments on the connectivity bottlenecks in the first stage are reported in Table 5.5.

The connectivity column describes how many convolutional neurons in the first feature extractor stage is connected to each of the input color channels. As can be seen, the Y:8 U:2 V:2 connectivity configuration as defined in the proposed architecture provides the best trade-off of accuracy against compute time.

As expected, a dense 12 neurons per channel connectivity, where all color channels are connected to all neurons in the first layer does produce higher accuracy, but at a cost of roughly 60-70% higher execution time.

For experimentation, special connectivity schemes were attempted also for the RGB color space, giving more connections to either the red or green channels exclusively, to compare against the Y channel preference on the YUV color space—but these results were not significant.

Execution time includes the pre-processing and all neural network executions.

Table 5.6: Results of sparse connectivity mapping between first and second convolutional stages.

Connectivity	**mAP**	**Execution Time (ms)**	
(Sparsity)		SoC	VPU
12 → 32 (1/1)	88.0	24.1	2.6
6 → 32 (1/2)	87.9	19.2	2.1
4 → 32 (1/3)	**87.4**	**17.4**	**1.9**
2 → 32 (1/6)	69.4	16.1	1.7
1 → 32 (1/12)	57.1	14.9	1.5
12 → 64 (1/1)	89.8	33.0	3.4
6 → 64 (1/2)	89.3	25.2	2.7
4 → 64 (1/2)	88.3	19.8	2.3
2 → 64 (1/6)	70.2	16.1	2.0
1 → 64 (1/12)	54.5	16.2	1.7

5.2.4 Second Stage Connectivity Mapping

Sparsity in the connectivity mapping between the first and second stage convolutional stages is summarized in Table 5.6. For all experiments, the first stage feature extractor maintained a number of 12 convolutional neurons, but only the number of connected neurons varied.

As can be seen, the 1/3 sparsity provides a good balance of execution time against accuracy. As in the previous experiment, the fully dense 1/1 factor provides a higher accuracy but at an increased computational cost.

It's interesting to see how accuracy quickly drops off in the least dense configuration with a 1/12 sparsity factor, but this is expected as a lot of the processing power in the neural network is lost if the connectivity is restricted too much.

Variations with the second feature extraction stage with a higher amount of neurons (64) was also tested to see the effects of such a design while maintaining similar sparsity limitations. With the exception of the most sparse configurations, it's as usual expected that a larger network configuration will have higher accuracy, but these effects were not very significant.

The execution times reported include both pre-processing and all neural networks executed.

Table 5.7

T_0	W	OC	Execution Time (ms)				Speedup
			Per-Window		Shared Map		
			SoC	VPU	SoC	VPU	SoC
4×4	6,596	98.4%	1,290	110	22	2.4	58.6x
8×8	**1,711**	**93.8%**	**340**	**37**	**17**	**1.9**	**20.0x**
12×12	772	85.9%	152	17	15	1.7	10.1x
16×16	459	75.0%	84	9.3	13	1.6	6.4x
20×20	284	60.9%	41	5.0	11	1.4	3.7x
24×24	215	43.8%	34	4.2	10	1.4	3.4x
28×28	151	23.4%	22	2.2	9.8	1.3	2.2x
32×32	130	0.00%	19	1.6	8.7	1.2	2.1x

5.2.5 Shared Maps

Table 5.7 shows results on different shared map window configurations. Experiments were conducted with several input layer stride T_0 configurations, from the closest packed 4×4 to the non-overlapping 32×32 layouts. A total window count W over the full 4 level pyramid, as well as the window overlap coverage OC per input map is given for each of the stride selections. The speedup factor is calculated showing the performance improvement of our method over the other in the SoC platform.

Choice of stride mainly affects the resolution of the localization of detected objects. The denser packed stride configurations will yield localization coordinates at discrete steps of 4 pixels, while the less packed non-overlapping configuration will result in coordinates that vary in steps of 32 pixels.

It is of great interest to note the final 32×32 configuration. Regardless of the fact that there is no overlap at this stride, a 2.1 speedup is still observed over running the windows individually. This is due to the inherent reduction in memory bandwidth through the system's pipelined execution approach, where the entire image needs to be loaded only once per execution. This contrasts the traditional approach where loading separate windows into memory at different times requires each to be individually sliced from the original memory block – a very expensive operation in the limited memory throughput of mobile devices.

Table 5.8: Results of various inference algorithms for result analysis after classification.

Algorithm	**mAP**	**Execution Time (ms)**	
		SoC	VPU
Maximum Value	80.3	0.4	0.3
Weighted Average	82.9	0.8	0.4
Neighbor Boosting	83.1	1.7	0.7
Energy Minimization	**87.4**	**2.2**	**0.7**

Table 5.9: Results of using different types of ensembles against a single neural network.

Networks	Ensemble Type	mAP	**Execution Time (ms)**	
			SoC	VPU
1-CNN	—	77.9	6.3	1.2
2-CNN	Same	78.7	10.9	1.4
2-CNN	Scaled	81.8	11.7	1.4
3-CNN	Same	79.4	16.9	1.9
3-CNN	**Scaled**	**87.4**	**17.4**	**1.9**
5-CNN	Same	79.9	25.3	2.4
5-CNN	Scaled	89.1	27.9	2.4

5.2.6 Energy Minimization by Belief Propagation

Table 5.8 gives an indicative comparison of the system against the competing techniques previously described in Section 3.5.

Execution time reported is for this algorithm only, not including neural network inference.

5.2.7 3-CNN Ensemble

Table 5.9 shows different ensemble configurations that were experimented with, ranging between 1 to 5 neural networks, and providing either same or scaled data.

Same data consists on training different networks with different parameters and

Table 5.10: Results of a comparison between the proposed object detection system and state of the art SSD-based object detection neural networks.

Networks	Ensemble Type	mAP	**Execution Time (ms)**	
			SoC	VPU
CNN	12C5 + 32C5	87.4	17.4	1.9
CNN	24C5 + 64C5	88.9	33.3	3.7
CNN	48C5 + 128C5	90.2	57.2	10.3
SSD	Mobilenet v2	96.3	390	45
SSD	Resnet 18	95.3	756	101
SSD	Resnet 32	97.2	1314	192

executing them all on the same input image patches. This is the traditional method of doing ensembles. Scaled data, as per the proposed system, consists on providing data at different scales, corresponding to the scale of data each network was trained with. This provides more context information and provides better protection to the network against false positives.

5.2.8 State of the Art Neural Networks

Table 5.10 provides the results for a really interesting experiment. Three variations of the proposed system are compared against state of the art SSD [35] object detection models with either a Resnet [40] or Mobilenet v2 [55] backbone.

These SSD networks are considered state of the art in object detection, and they certainly outmatch the accuracy of the proposed system. Especially since the data complexity of the benchmark application is considerably lower than what these models are capable to work with. However, even the Mobilenet v2 backbone has a considerably high inference time, which on a mobile smartphone SoC makes impractical for continuous real time operation.

The proposed system, on the other hand, even though the mAP metrics can not match what SSD achieves, the accuracy is still competitive, while the execution time is an order of magnitude lower.

Note: The proposed system executes on the platforms with the implementation detailed in Chapter 4, while the SSD models execute with Tensorflow Lite on the SoC and OpenVino on the VPU.

5.3 Discussion

This chapter has reviewed several experiments performed with the proposed object detection system. The various design decisions described in Chapter 3 have been reviewed through a series of ablation tests. Some of these decisions have proven to be the most optimal configuration, while others are chosen as a matter of trade-off between achieving a certain level of accuracy while maintaining the execution time low.

The same applies to the final experiment present comparing against higher complexity models. These results present the spirit with which this work is put forth. The system developed in this work is extremely efficient and fast in execution time.

It is important to note that the execution time is proportional to energy consumption, which is an even more important metric in edge computing devices. Obviously camera systems run at slower frame rates than what this system can execute at. But the fact that the detection system completes its cycle in a short period of time results in finite power sources to last for longer, extending the usable lifespan of portable devices that might be executing such a system. Furthermore, a system running under low loads like this is far less likely to throttle due to thermal conditions, which is a frequent occurrence in mobile SoCs that rely only on passive cooling techniques.

6 Application Use Case

To demonstrate how the proposed system can be deployed in real world applications, a sample use case has been developed based on the efficient object detection network proposed in this work. This application attempts to resolve the problem of ear detection as an alternative to traditional biometric algorithms.

6.1 Introduction

The problem of people recognition by means of identifying them biometrically by their ear has received considerable attention in the literature [56–59]. However, for an ear recognition system to be accurate, the first and obvious step it must take is to properly detect the presence and location of an ear within an image frame. This seemingly simple task is often made more difficult because in practice, such images very commonly present the subject's ear in poses which are much different to those a system is usually trained for. Furthermore, occlusion and partially visible ears is very common in natural images, and it presents a challenge which must be addressed.

An issue to consider is the great importance of robustness against pose variation and occlusion when an ear detection algorithm is put to practice. It is worthwhile to note that most of the detection systems listed above are not tested nor developed for difficult occlusion scenarios, such as partial occlusion by the hair, jewelry, or even hats and other accessories. The most likely reason is simply the lack of public datasets containing appropriately occluded images. Furthermore, to the best of our knowledge, there is no major research that has been performed on the effect of ear occlusion in natural images.

Table 6.1: Details on the contents of the various datasets used in this work.

Dataset	Dataset Size	Subjects	Images per Subject	Resolution Size *pixels*	Content	Source
AMI [60]	700	100	7	492×702	Closeup ears, both sides	*Photo*
UND [61, 62]	464	114	4	1200×1600	Bust profile, right side only	*Photo*
Videos (Train)	950	5	190	1920×1080	Head profile, both sides	*Video*
Videos (Test)	910	7	130	1920×1080	Head profile, both sides	*Video*
UBEAR v1.0 [63] (Train)	4497	127	35	1280×960	Head profile, both sides, and masks	*Video*
UBEAR v1.1 [63] (Test)	4624	115	40	1280×960	Head profile, both sides	*Video*

6.2 System Description

6.2.1 Datasets

The existence of ear-centric data is limited and sparse. There exist no standard datasets upon which a large body of work can be contrasted with. In this work, however, we attempt to use a variety of datasets in order to establish some benchmarks upon which future works can be built upon. For this purpose, we use a total of four datasets in our experiments. Three of these are public and only one is private. Each of these datasets has a set of features which make them particularly useful for a particular task, and each one introduces new challenges. As such, we use them all to base a selection of real-world experiments on each. Table 6.1 gives an overview of the content in each dataset.

The first dataset is the AMI dataset [60], a collection of 700 closeup images of ears. These are all high quality images of ears perfectly aligned and centered in the image frame, as well as having high photographic quality, in good illumination conditions and all in good focus. This dataset is therefore exemplary in order to test the recognition sensitivity towards different ears, however, due to the closeup nature of the images, they are not really well suited for ear localization tasks.

The second dataset used is the UND dataset [61, 62]. A collection of photographs of multiple subjects in profile, where the ear covers only a small portion of the image.

The photographic quality of these images is very high, and again all in constant and good illumination, and with none of the ears being occluded by hair or other objects. The poses of subjects varies very slightly in relation to the camera, but not so much as to introduce distracting effects due to head rotation and pose. As a result, these images are suitable in testing the specific task of localization among a large image frame, while avoiding the challenges of viewpoint and illumination variation.

The third dataset is the Video dataset. A private collection of 940 images composed of HD frames extracted from short video sequences of voluntary participants. There are 14 image sequences of 7 subjects—one for each person's ear. Each sequence consists of 65 frames from a span of approximately 15 seconds in time extracted from a continuous video. The subjects were asked to rotate their heads in various natural poses following smooth and continuous motions throughout the sequence. The illumination and environment are relatively consistent across all videos, and subjects were asked to move any potential occlusion away from their ears. We use this dataset primarily to test the detector sensitivity only towards different relative rotations of the subject's head in relation to the camera, while avoiding challenges due to variable illumination. The higher number of images per subject, combined with a low number of total subjects, are useful to also reduce the effect from using a large number of wildly variable ear shapes in the tests, and again, concentrate mainly on their pose. A variation of this dataset was created and set aside for training purposes. This comprised profile image frames from an additional 5 participants, different from the subjects in the test dataset.

The final and perhaps most important dataset we use is the UBEAR [63] dataset. This is a very large collection of images of subjects shot under a wide array of variations, which spans multiple dimensions—not only in pose and rotation, but also in illumination, occlusion, and even camera focus. These images, therefore, simulate to a very good degree the conditions of photographs in non-cooperative environments were natural images of people would be captured ad hoc and used to carry out such a detection. These images, although definitely being ear-centric, make no attempt at framing or capturing the ear under perfect conditions, and as such reflect a real-world test scenario. As our main interest in this work is the detection of ears in natural images, this then becomes our main dataset to test the fullest potential of the system we propose.

6.2.2 Convolutional Neural Network

The target use case of the system is to perform real time ear detection, especially with input video streams. For this, a system that can run quickly is a fundamental requirement. For this reason, an optimized architecture is needed. The target classes we seek to recognize with the neural network are only three: (i) Left Ear, (ii) Right Ear, and (iii) Background—referred to by their corresponding abbreviations: LE, RE, and BG in all the following descriptions of the system. As the data variability within each class is relatively low, with many training data samples having a similar set of characteristic ear features, the network can perform relatively well by learning only a small number of unique features (unlike the case of large modern CNNs). Therefore, a small neural network, with a low layer and neuron count is enough to learn the training data used by this system.

Furthermore, a size of 64×64 is selected for the input data of the network, as images at this size carry enough features and information to properly define the ear shape, while at the same time not being so large that the system would require large convolutional kernels to properly analyze the images.

Finally, as shared maps execution will be used to do the analysis over full images, the maximum accumulated pooling factor needs to be kept small. This ensures that the stride size on the final output map is still small for fine localization to take place. For this reason, 3 convolutional and pooling layers are decided as the base of the architecture.

Knowing these three constraints, for the input and output, and the maximum number of layers, through a process of iterative trial and error, a final architecture was decided upon as follows:

$$18C5 + MP3 + 36C5 + MP2 + 36C5 + MP2 + 144L + 3L$$

6.2.3 Training Data

Through a 3-CNN ensemble method, the system will comprise three individual neural networks, we already know beforehand that the training data will need to be gathered in accordance to the requirements for each of the individual networks. Each network will essentially analyze three different crop sizes of each region, so the data for all three can be prepared simultaneously by simply starting with one

dataset, and extending it by cropping and scaling accordingly to generate the data for the two other sizes.

Existing image datasets consisting of annotated ear photographs are very scarce and small in volume. Creating sufficiently large amounts of training data, therefore, required a lot of manual labor in image manipulation. The dataset UBEAR v1.0 was particularly helpful, as described in Section 6.2.1, in that it includes for each of its images a ground truth mask. This mask outlines the exact location of the ear in each image, and this aided in cropping out the corresponding bounding boxes for each ear. Not all patches from this dataset could be used, however, as many were extremely blurry and not appropriate for training. In the end, approximately 3,000 images were used from this set for training data.

Furthermore, we supplemented the training data with additional samples that were manually cropped from video frames. These originate from the Videos (Train) dataset described in Section 6.2.1. With this addition, the training dataset now consisted of roughly 4,000 images.

To increase the dataset size even more, the data was augmented in two ways: (i) images were randomly modified by adding small translations, rotations and rescales; and (ii) images were horizontally flipped, and the resulting image was assigned to the opposite ear dataset. This artificial augmentation boosted the training data size tenfold. It now consisted of approximately 40,000 images, or 20,000 for each ear.

In order to prepare for the training of our final 3-CNN architecture, we processed the images for each ear side into three separate sets, for each of the 3-CNN scales: S, M, and L. This was done by simply cropping and rescaling each sample appropriately.

The process was repeated for both sides, thus producing six separate image collections for left and right ears, at each of the three scales. Finally, one more background noise dataset was also created, of the same size as the others, and consisting of randomly cropped patches from a large flickr photo database and from non-ear regions of the UBEAR and Videos training sets.

In total, we ended up with seven distinct collections for training purposes, each one consisting of roughly 20,000 images. Figure 6.1 shows an example of these.

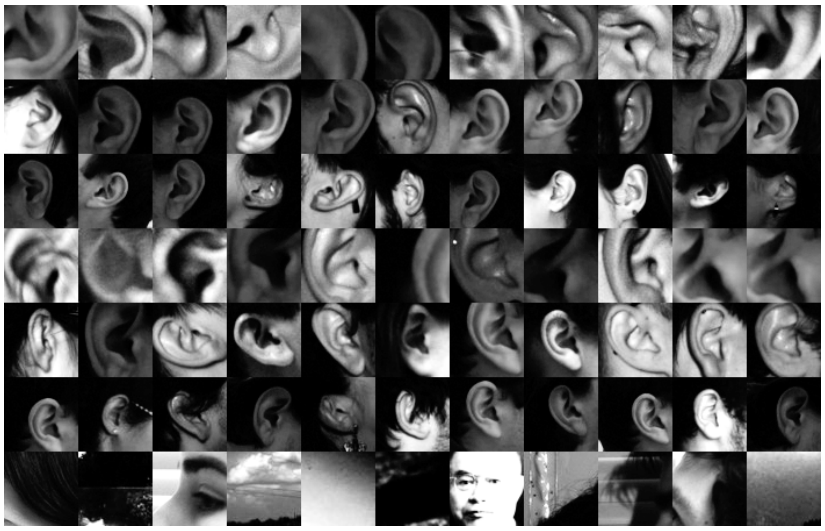


Figure 6.1: [
Training Dataset Subset]A small subset of each of the seven datasets used for training. From top to bottom: Left-Small, Left-Medium, Left-Large, Right-Small, Right-Medium, Right-Large, Background, prepared in YUV color space.

6.2.4 Network Training

Our final neural network classifier was trained with the three-scale collection described above. Each of the three networks used a 3-class training dataset compiled from left and right ears at the corresponding network scale, and a copy of the background image collection.

The structure of all three networks was exactly the same, and is the one described in Section 6.2.2. The input consists of an image resized to a square of size 64×64 in the YUV color space. The input images are then passed through a pre-processing step which consists of Contrast Normalization process, which helps to enhance image edges and redistribute the mean value and data range.

Each network is trained with its corresponding small, medium or large datasets. A standard SGD approach was used for training, and ran for a duration of approximately 24 iterations until no further improvement could be made on the test-fold of the data. Ideal targets for each of the output labels were assigned in the $[-1, +1]$ range, where active labels are positive, and inactive labels are negative. This distribution was chosen in this manner (as opposed to the more traditional $[0, 1]$ range) to aid with the 3-CNN inference.

Table 6.2: Final confusion matrix of the training data fold.

Classified As / Real Class	Left Ear	Right Ear	Background	Total in Class	Accuracy (%)
Left Ear	16,040	56	88	16,184	99.11
Right Ear	46	16,064	74	16,184	99.26
Background	63	194	15,927	16,184	98.41
			Total	48,552	98.93

Table 6.3: Final confusion matrix of the testing data fold.

Classified As / Real Class	Left Ear	Right Ear	Background	Total in Class	Accuracy (%)
Left Ear	3964	34	49	4047	97.95
Right Ear	14	4002	31	4047	98.89
Background	8	42	3997	4047	98.77
			Total	12,141	98.54

All datasets are divided into training and testing folds, at an 80% to 20% ratio as per standard machine learning training practices. The final results of training over these two sets are summarized in Tables 6.2 and 6.3.

6.2.5 Detection

Runtime operation of the network is performed through shared map execution of CNNs. This allows for an optimized method of inferring detection predictions from a full image frame in a manner that is much more efficient than the traditional sliding window approach.

The process requires the input image to be first prepared as a multi-scale pyramid. This is simply to be able to detect ears in all possible sizes relative to the image frame, so as to be able to properly carry out the detection, regardless of the subject's relative distance to the camera.

Each of these pyramid levels will be given to each of the three networks to be

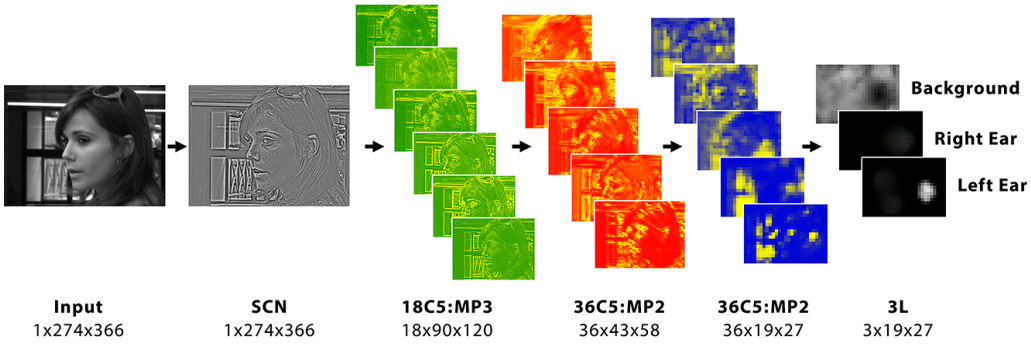


Figure 6.2: [
3-CNN Layout]Shared map execution of one of the CNNs over a sample input image.

analyzed independently. Each network, thus, creates three output maps per level, corresponding to each of the target classes trained, LE, RE, and BG. Figure 6.2 depicts the shared map execution of one of the networks for a particular pyramid level of size 274×366.

Every pixel in each of these output maps corresponds to that class' predicted likelihood at a window whose location can be traced back to the input image according to the shared map's alignment and position configuration. Figure 6.3 shows how windows can be re-constructed from these shared maps and they correspond precisely to the multiple detections that a traditional sliding window approach would produce, but at a fraction of the computing time. These multiple detections can be grouped together via Non-Maximum Suppression [50] with an IOU of 0.5.

A summary of this whole process is listed below:

for all $P \in \{PyramidScales\}$ **do**

$Image_{CN} = CoNorm(Image_P)$

for all $Z \in \{L, M, S\}$ **do**

$O_Z^{LE}, O_Z^{RE}, O_Z^{BG} \leftarrow LBP(SharedMap(Image_{CN}, Network_Z))$

end for

for all $K \in \{LE, RE, BG\}$ **do**

$O_P^K \leftarrow Ensemble(O_S^K, O_M^K, O_L^K)$

end for

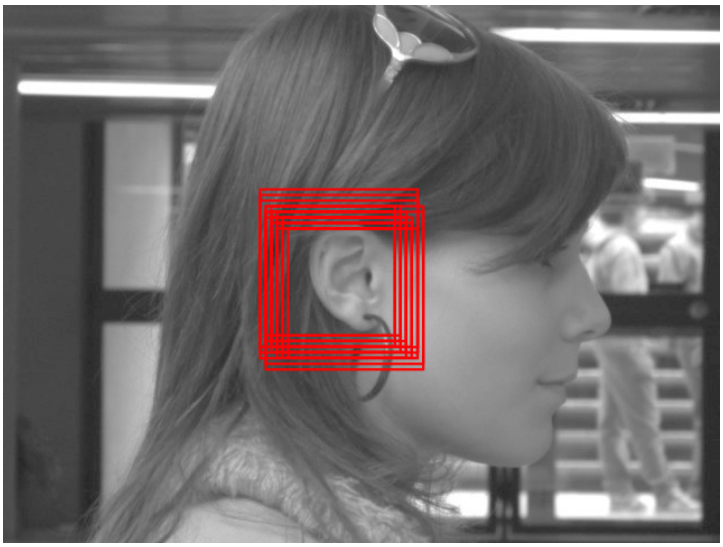


Figure 6.3: Sample of multiple overlapping detections casted as individual detection windows on an input image.

end for

for all $K \in \{LE, RE\}$ **do**

$G^K \leftarrow NMS(O_P^K)$

if $G^K > Threshold$ **then**

$Keep(G^K)$

else

$Discard(G^K)$

end if

end for

The correct threshold to use should be carefully decided upon depending on the type of data being analyzed. In the case of the AMI database, where images are already prepared as cropped ears, the system detects no False Positives whatsoever, and thus the threshold value decision does not affect the False Positive rate in any way. In this case, a very low (or zero) threshold can be chosen in order to maximize the number of correctly detected ears. This can be seen in the results shown in

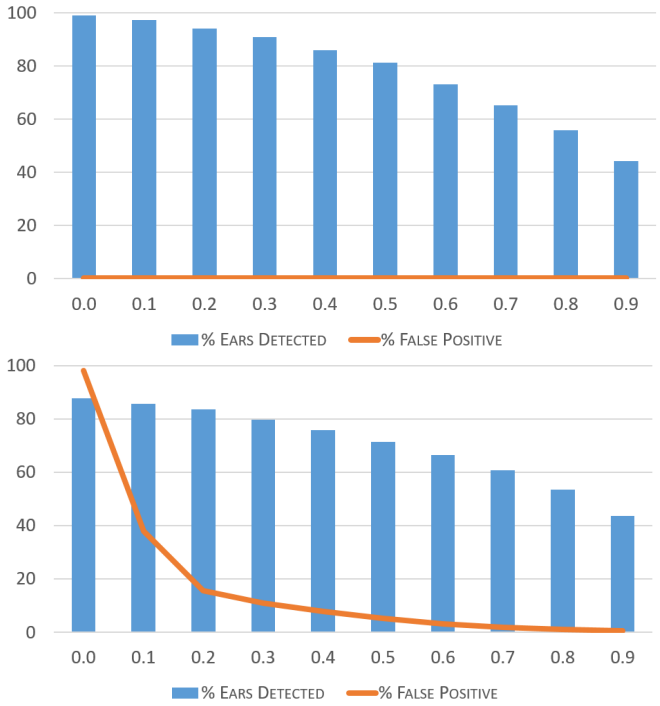


Figure 6.4: Threshold sensitivity on ear detections: (**Top**) AMI Dataset Detections, (**Bottom**) UBEAR Dataset Detections.

Figure 6.4 (Top), where the accuracy rate of varying threshold amounts is depicted.

In the case of natural images in non-cooperative environments as with the UBEAR dataset, the effect of false positives is much more important, as can be seen in Figure 6.4 (Bottom), where small variations in the threshold value lead to a drastic drop in the false positive rate, while not significantly affecting the accuracy of detected ears.

6.3 Experiments

6.3.1 Test Methodology

Multiple experiments were conducted with the various datasets in order to evaluate the system's accuracy in different scenarios. For all tests, the experiment was carried out with the 3-CNN method proposed in this work. To contrast the results, the same tests were also performed with a standard Haar Cascade Classifier trained

on similar data as implemented in OpenCV [64], and executed with a similar sliding window configuration while post-processing them with the same window grouping algorithm.

In all cases, the results reported are defined as follows:

- **True Positive:** Detection groups which successfully enclose the bounding box of an ear within the image.
- **False Positive:** Detection groups which mis-classify the side of the ear detected, or which erroneously detect noise in the image that does not correspond to an actual ear.
- **False Negative:** Ears in an image which failed to be detected by the network entirely, or whose final detection group confidence value was below the selected threshold.
- **True Negative:** This value would usually describe the rate at which non-ear noise is successfully ignored by the classifier. However, in the case of full image frames, this would greatly offset the result bias by greatly increasing the overall classification accuracy needlessly. We avoid recording this on purpose such that the results given represent the true nature of correctly classified ears only.

The performance metrics reported for all cases are the precision which measures the exactness of the classifier; the recall which measures its completeness; and the F1 metric which provides a balance between precision and recall, and is therefore a more objective comparison of the performance of two classifiers. Furthermore, the traditional accuracy rate is also reported, in order to provide a basic performance metric.

6.3.2 Video Analysis

We test the detection accuracy on individual video frames. An experiment was carried out with the Video dataset as described in Section 6.2.1. The purpose of this test is to ensure that both ears can be correctly classified as either left or right, while working with data of variable head poses.

Results of these tests is presented in Table 6.4, where it can be seen that our system greatly outperforms Haar in this particular task.

Table 6.4: Results of testing over the Videos dataset.

		Haar				3-CNN			
	Subset Size	**Precision (%)**	**Recall (%)**	**Accuracy (%)**	**F1 Score (%)**	**Precision (%)**	**Recall (%)**	**Accuracy (%)**	**F1 Score (%)**
Middle	470	97.60	97.60	95.32	97.60	99.57	99.79	99.36	99.68
Upwards	162	100	69.75	69.75	82.18	95.95	91.03	87.65	93.42
Downwards	284	98.77	57.09	56.69	72.36	94.83	95.19	90.49	95.01
Left Ear	455	97.85	71.21	70.11	82.43	97.07	97.29	94.51	97.18
Right Ear	461	98.53	88.57	87.42	93.29	97.98	96.46	94.58	97.21
Complete Dataset	**916**	**99.05**	**80.07**	**79.45**	**88.55**	**97.59**	**96.95**	**94.68**	**97.27**

The significance of this test is in the ability to continuously detect the same ear on a moving image sequence, regardless of head orientation. The high detection rate ensures that the ear is consistently detected during the majority of each video's duration, except for a few odd frames where detection might fail from time to time. However, a few frames later, the ear is found again and detection continues as normal. This result rate would therefore allow for a tracking mechanism to be successfully implemented in such video streams.

6.3.3 Image Resolution

Detecting images of subjects at a great distance from the camera is usually problematic. To quantitatively measure the performance of the system in cases where the relative size of the image is very small, various tests were performed on the AMI dataset with the ears previously resized at different scales, ranging from 16×16 up to 96×96. The results of both the combined 3-CNN system as well as that of the individual S, M, and L CNNs are displayed in Figure 6.5 (Top).

This shows that even ears which are found at scales much lower than the networks' input size of 64×64 can still be successfully detected, albeit at a lower rate depending on the actual size.

Figure 6.5 (Bottom), in particular, explains the dropoff in resolving power at smaller scales. The S CNN is the first one to fail at diminishing scales, as could be expected due to the nature of the data this network analyzes. Meanwhile, the other two CNNs continue to detect with sufficient accuracy at even the smallest scales. Arguably, it could be said that a system without the S scale might do better for this particular purpose, as the dropoff exhibited by the S CNN is the main reason behind the 3-CNN difficulty in detecting smaller sized ears. However,

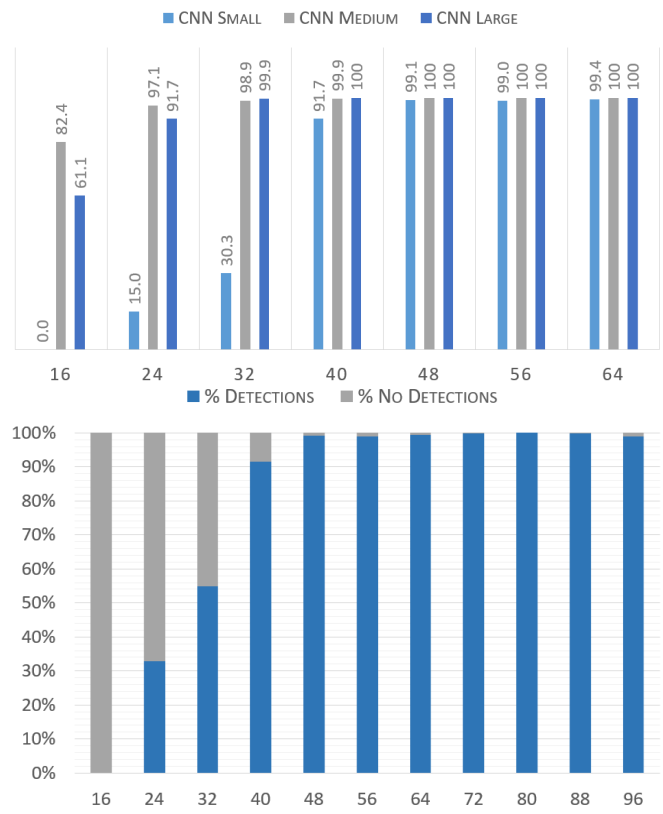


Figure 6.5: Image resolution and ear size sensitivity: (**Top**) Individual CNNs, (**Bottom**) 3-CNN System.

the S CNN has been shown before to be essential for noise differentiation, and as such, this side effect is an acceptable trade-off.

6.3.4 Non-cooperative Natural Images

Traditional computer vision approaches usually require the ear to be perfectly aligned, or at the very least in the same plane as the photograph projection, thus imposing restrictions that are very restrictive when analyzing real world imagery. Due to the ability of CNNs to learn multiple representations of the same object, and given the pose variety used in the training data, the final trained system is capable of detecting ears at very different angles with respect to the camera.

The UBEAR dataset contains labels for each image which facilitates its partitioning

according to the relative pose of the subject in relation to the camera. Tests were run over the full dataset and the results were divided according to the angle of the subject's gaze. These results are depicted in Figure 6.6 and summarized in Table 6.5.

The common trend of our 3-CNN outperforming Haar continues to be seen here. However, the real significance behind these results is that Haar, not unlike most traditional computer vision approaches, is highly dependent on viewpoint, and its performance largely drops off as the angle varies from the more normal "Middle" and "Towards" angles. Meanwhile, our 3-CNN system maintains a very similar and stable performance rating regardless of the angle at which the ear is presented.

Further UBEAR labels can be used to split the data into additional folds, such as ear sidedness. As expected, the system works mostly the same for either left or right side ears. The small differences in the results might just be due to a random variation in the images, and not to a real side preference of the classifier.

Finally, we tested the system on images which were marked to have occlusion against those that did not. Occlusion is not a defined label in the UBEAR dataset, therefore, for this study, we manually defined this data fold based on a subjective decision of which images could be considered as occluded. This is because degrees of occlusion can vary from merely a few small strands of hair or a small earring, to very large accessories or full sections of hair covering well over half of the ear. The final *occlusion threshold* decision was made to mark only those ears which had their outline covered at least 25%. This resulted in approximately one third of the images to be marked as occluded.

Not surprisingly, the 3-CNN system performs better when no occlusion is present. However, it is worth noting that even when analyzing occluded ears, the 3-CNN system outperforms Haar when it analyzes clearly visible, and non-occluded ears.

A final study was performed on gender sensitivity of the detector. The classifiers are not necessarily sensitive to the different shapes of male and female ears. However, a visible disparity can be seen, simply due to the fact that female ears are far more likely to be occluded by longer hair or more prevalent accessories such as large earrings. Thus, gender sensitivity results closely resemble those of occlusion sensitivity.

Figure 6.7 shows a few selected samples of the 3-CNN and its detection in particularly challenging images, due to either occlusion or extreme viewpoint perspectives.

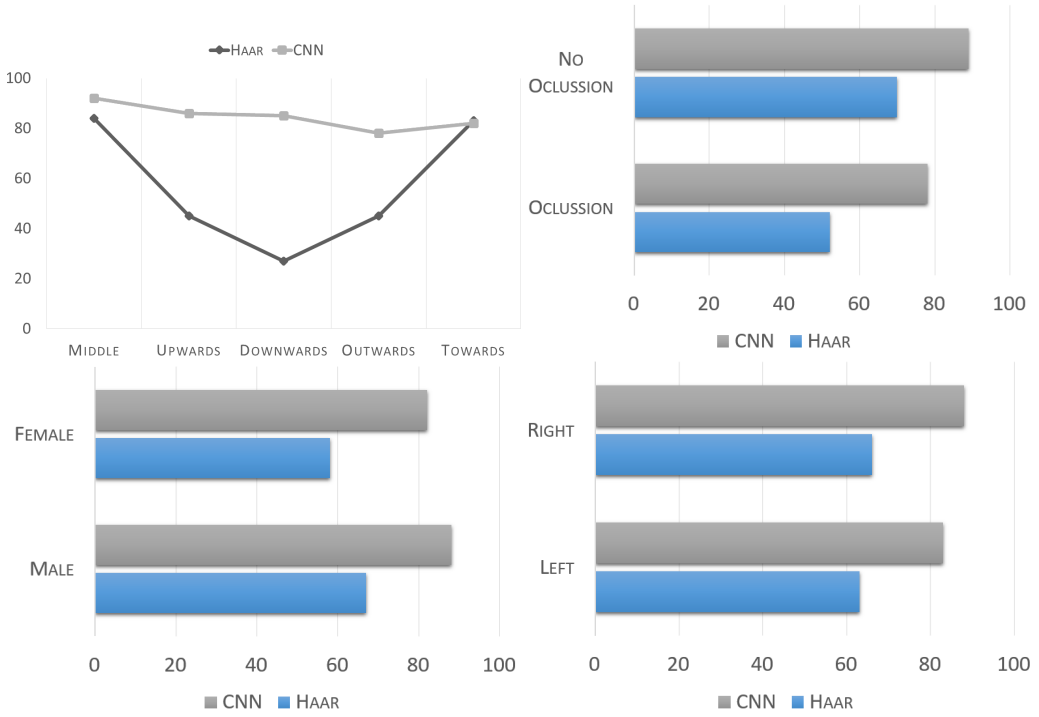


Figure 6.6: Detection performance of our 3-CNN system vs Haar on the different data folds of the UBEAR dataset: (**Top Left**) Angle Sensitivity, (**Top Right**) Occlusion Sensitivity, (**Bottom Left**) Gender Sensitivity, (**Bottom Right**) Ear Side Sensitivity.

Table 6.5: Detection performance of our 3-CNN system vs Haar on the different data folds of the UBEAR dataset, grouped by: Angle, Gender, Ear Side, Occlussion, and Total.

		Haar				**3-CNN**			
	Subset Size	**Precision (%)**	**Recall (%)**	**Accuracy (%)**	**F1 Score (%)**	**Precision (%)**	**Recall (%)**	**Accuracy (%)**	**F1 Score (%)**
Middle	1392	95.90	74.31	72.03	83.74	89.89	93.92	84.95	91.86
Upwards	813	85.87	30.42	28.97	44.93	87.47	84.94	75.73	86.19
Downwards	784	88.65	16.23	15.90	27.44	85.68	84.57	74.09	85.12
Outwards	789	89.96	30.34	29.35	45.37	85.92	71.26	63.81	77.91
Towards	829	95.10	73.24	70.57	82.75	79.70	84.40	69.47	81.99
Male	3403	94.17	51.58	49.99	66.65	86.10	87.84	76.93	86.96
Female	1204	91.06	42.47	40.77	57.92	86.99	77.83	69.71	82.15
Left Ear	2289	93.49	47.33	45.82	62.84	83.64	83.11	71.48	83.37
Right Ear	2318	93.42	51.07	49.30	66.04	88.97	87.32	78.79	88.14
Occlusion	1491	89.70	36.49	35.03	51.88	85.01	71.63	63.60	77.75
No Occlusion	3116	94.70	55.24	53.59	69.78	86.79	91.53	80.34	89.10
Complete Dataset	**4607**	**93.45**	**49.22**	**47.58**	**64.48**	**86.31**	**85.23**	**75.08**	**85.77**

Figure 6.7: Sample detections on particularly difficult images from the UBEAR dataset, including extreme head orientations and occlusion.

6.3.5 Summary

To conclude, Table 6.6 lists a summary of all total results across all four datasets while comparing our 3-CNN system with the well known Haar Cascade Classifier algorithm.

As can be seen, the CNN based system always outperforms the Haar algorithms in all sets, by an amount ranging between 10% to 29% in the F1 metric. This is particularly so in the UBEAR dataset, since the Haar classifier is incapable of modelling the higher variety of internal representations required to properly classify images in that dataset.

Figure 6.8 shows a summary of these results. It is important to remark that that our proposed system has stable performance figures across the first three datasets, all of which consist of perfect purpose-made ear photography. The results only slightly drop when presented with natural images due to the challenges already described. This is in contrast to the Haar classifier, which has wildly disparate results, demonstrating the large dependency of this system on the particular conditions of one dataset or another.

Table 6.6: Summary of the total results over all four datasets contrasting the Haar and 3-CNN algorithms.

Dataset	Algorithm		Positive	Negative	Precision (%)	Recall (%)	Accuracy (%)	F1 Score (%)
UND [61, 62]	3-CNN	**Positive**	461	20	95.84	99.35	95.25	97.57
		Negative	3	0				
	Haar	**Positive**	270	7	97.47	58.44	57.57	73.07
		Negative	192	0				
Videos	3-CNN	**Positive**	890	22	97.59	96.95	94.68	97.27
		Negative	28	0				
	Haar	**Positive**	727	7	99.05	80.07	77.47	87.31
		Negative	181	0				
AMI [60]	3-CNN	**Positive**	693	0	100.00	99.00	99.00	99.50
		Negative	7	0				
	Haar	**Positive**	382	7	98.20	55.12	54.57	70.61
		Negative	311	0				
UBEAR [63]	3-CNN	**Positive**	3814	605	86.31	85.23	75.08	85.77
		Negative	661	0				
	Haar	**Positive**	2227	156	93.45	49.22	47.58	64.48
		Negative	2298	0				

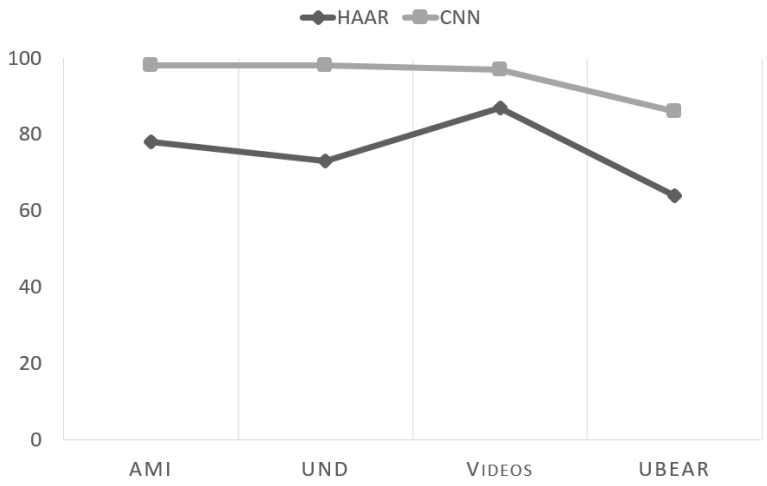


Figure 6.8: Results of our 3-CNN system compared to the Haar classifier over the various test datasets.

7 Conclusions

In this work, an efficient object detection system based on deep convolutional neural networks was presented. All the design decisions involved in the development of this system were reviewed in full detail, although there were some subtleties in the implementation and programming details which for brevity were not mentioned here.

A method for pre-processing image data was described, which treats the image through the YUV color space as given by ISPs in mobile systems. The algorithm takes advantage of design elements such as color information compression and the availability of an optimized convolutional operation that can execute efficiently in hardware. A neural network backbone architecture as described which uses a minimal amount of connections and neurons to produce classification inference results. This network was adapted and extended spatially in a method called shared maps such as to transform the classification-only ability of the backbone to a classification and localization capability which are essential cornerstones of object detection. The final element in the proposed architecture was a discrete inference algorithm which uses energy minimization as a method to reduce the active detections and find the true location of an object.

A review for implementing efficiently this neural network system was also provided, given the commonly available hardware compute units present in standard edge computing devices. The hardware and compute characteristics of different compute elements were reviewed, as well as their capabilities and a description of how they can be programmed from software with optimization in mind to perform neural network inference.

This thesis has demonstrated that it is possible to develop a highly efficient CNN-based object detection system that is capable of running in constrained environments. Compared to state of the art detection systems, this architecture provides limited, but acceptable accuracy at several orders of magnitude faster execution time, a measure that translates to expended energy in edge computing devices.

The work presented here opens up many interesting possibilities regarding the types of computer vision systems that can be deployed in the edge. While this architecture may not be powerful enough to drive mission critical decision making systems, it does provide a solution for mass detection in large quantity deployments. It also provides a framework upon which to build further and continue developing more capable systems.

As technology continues to advance and embedded devices continue becoming more economical, it is naturally possible to continue scaling up the size and complexity of the systems that can be deployed on them. However, many of the optimization considerations established in this work will continue having an impact on future generations of mobile computer vision deployments.

As future lines of research it would be of great interest to study the efficient implementation of other types of neural network systems, for example recurrent neural networks capable of analyzing sequences of images for tasks such as behavior prediction.

To conclude, this has proven to be an interesting research project, but this is only the beginning of what is surely to be a fruitful field to continue working on.

A Convolutional Layer Kernel

A full version of the code of a Renderscript kernel. This kernel makes use of all the optimization techniques described in this work, including:

- Loop unrolling is applied for the inner loops of convolution.
- Kernel specialization is applied by fixing this sample kernel to a particular configuration.
- Memory Access Patterns are maintained in row-major format for all inputs and outputs.
- Layer Fusion is achieved by fusing the convolutional layer with the max-pooling layer.
- Efficient Configuration Data used when reading the kernel weight values.
- Approximation of non-linearity functions by a piece-wise linear function.

Listing A.1: **macros.rsh** Renderscript header file with some commonly used macros.

```
1  #pragma version(1)
2  #pragma rs java_package_name(
3          eu.fsmc.convis.compute.renderscript.process
4  )
5
6  #define IK(ip, kp) input[ip] * kernel[kp]
7
8  #define CONVOLUTION_3x3_ROW(ip, kp) \
9          IK(ip - 1, kp - 1) + \
10         IK(ip    , kp    ) + \
11         IK(ip + 1, kp + 1)
```

```
12
13 #define CONVOLUTION_5x5_ROW(ip, kp) \
14         IK(ip - 2, kp - 2) + \
15         IK(ip - 1, kp - 1) + \
16         IK(ip    , kp    ) + \
17         IK(ip + 1, kp + 1) + \
18         IK(ip + 2, kp + 2)
19
20 #define CONVOLUTION_3x3(ipa, ip, ipb) \
21         CONVOLUTION_3x3_ROW(ipa, 1) + \
22         CONVOLUTION_3x3_ROW(ip , 4) + \
23         CONVOLUTION_3x3_ROW(ipb, 7)
24
25 #define CONVOLUTION_5x5(ipaa, ipa, ip, ipb, ipbb) \
26         CONVOLUTION_5x5_ROW(ipaa,  2) + \
27         CONVOLUTION_5x5_ROW(ipa ,  7) + \
28         CONVOLUTION_5x5_ROW(ip  , 12) + \
29         CONVOLUTION_5x5_ROW(ipb , 17) + \
30         CONVOLUTION_5x5_ROW(ipbb, 22)
31
32 #define MAXPOOLING_2x2(p1, p2, p3, p4) \
33         max(max(max(p1, p2), p3), p4)
34
35 #define TANH_APPROX(x) \
36         (x < -1 ? -1 : (x > 1 ? 1 : x))
37
38 #define TANH_EXACT(x) \
39         tanh(x)
40
41 #define MAP(om, im) mapping[im + om * mapInputs]
42
43 }
```

Listing A.2: **c5mp2fused.rs** Renderscript kernel for 5×5 convolutional layer fused with a 2 × 2 max-pooling layer.

```
int *mapping;
int mapInputs;
int mapOutputs;

float *input;
int inputWidth;
int inputHeight;
int inputMapWidth;
int inputMapHeight;

float *bias;

float *kernels;
int kernelSize;
int paddingSize;

float *output;
int outputWidth;
int outputHeight;
int outputMapWidth;
int outputMapHeight;

void root(const int32_t *v_in, int32_t *v_out,
          uint32_t x, uint32_t y) {
  int om = y / outputMapHeight;
  int my = y % outputMapHeight;

  int pm = x / outputMapWidth;
  int mx = x % outputMapWidth;

  int ix = pm * inputMapWidth + paddingSize + mx * 2;

  float tl = bias[om];
  float tr = bias[om];
  float bl = bias[om];
```

```
36    float br = bias[om];
37
38    for (int im = 0; im < mapInputs; im++) {
39      int inputMap = MAP(om, im);
40      int iy = inputMap * inputMapHeight +
41               paddingSize + my * 2;
42
43      int ip   = ix + iy * inputWidth;
44      int ipa  = ip   - inputWidth;
45      int ipaa = ipa  - inputWidth;
46      int ipb  = ip   + inputWidth;
47      int ipbb = ipb  + inputWidth;
48      int ipbn = ipbb + inputWidth;
49
50      float *kernel = kernels + (om * mapInputs + im) *
51                      (kernelSize * kernelSize);
52
53      tl += CONVOLUTION_5x5(
54            ipaa    , ipa    , ip     , ipb     , ipbb
55      );
56      tr += CONVOLUTION_5x5(
57            ipaa + 1, ipa + 1, ip  + 1, ipb  + 1, ipbb + 1
58      );
59      bl += CONVOLUTION_5x5(
60            ipa     , ip     , ipb    , ipbb    , ipbn
61      );
62      br += CONVOLUTION_5x5(
63            ipa  + 1, ip  + 1, ipb + 1, ipbb + 1, ipbn + 1
64      );
65    }
66
67    int op = x + y * outputWidth;
68
69    float res = MAXPOOLING_2x2(tl, tr, bl, br);
70
71    output[op] = TANH_APX(res);
72  }
```

B Summary in Spanish

This Appendix provides a summary in the Spanish language of the key contents of this thesis.

Introducción

Los sistemas visuales biológicos son capaces de reconocer objetos fácilmente debido a una gran cantidad de especialización y adaptación a través de la evolución natural. Estos sistemas están interconectados con otras partes del sistema nervioso, como los centros de memoria y razonamiento en el cerebro, por lo que la tarea de visión no es algo que pueda separarse y resolverse de manera independiente. Sin embargo, esto es lo que los sistemas de visión artificial tratan de lograr hoy en día, principalmente porque las sutiles complejidades de la visión biológica son extremadamente difíciles de reproducir. Sin embargo, todavía es posible inspirarse en cómo funcionan ciertos aspectos de la visión natural, para imitar funciones sencillas.

Inicialmente, los sistemas de visión artificial adoptaron un enfoque sintético para resolver el problema. Los algoritmos de reconocimiento de imágenes como SIFT [1], SURF [2] o HOG [3] se basan en descriptores de características ajustadas manualmente, y aunque están lejanamente basados en principios biológicos, como la sensibilidad a ciertas orientaciones, estos utilizan principalmente las cualidades geométricas de la imagen y fallan fácilmente a medida que incrementa la complejidad de las formas. En los últimos años, sin embargo, se ha hecho evidente que es necesario una solución mejor. Uno de los algoritmos más exitosos han sido las redes neuronales profundas inspiradas biológicamente [4], donde las imágenes de entrada se procesan a través de varias capas de extractores de características y finalmente se reconocen como pertenecientes a una de varias clases posibles. Estos trabajos se basan en los hallazgos de Hubel y Wiesel [5] donde se identificaron dos tipos de células dominantes en la corteza visual primaria de los animales mamíferos, las

llamadas células simples y células complejas, conectadas a través de un modelo jerárquico en cascada. Inspirado por estos hallazgos, el Neocognitron introducido por Fukushima [6], hace uso de dos tipos de células que asumen de manera similar los roles de extractores de características locales para lograr tolerancias contra la deformación.

Más adelante, LeCun [7] introdujo la red neuronal convolucional (CNN por sus siglas en inglés) como un sistema más robusto para la identificación de caracteres escritos a mano en documentos escaneados. Nuevamente, se utiliza un sistema similar con dos tipos de células, donde una capa convolucional extrae características y una capa de submuestreo reduce el la dimensionalidad de la imagen para tolerar las variaciones geométricas de estas características. Estas redes neuronales convolucionales forman la base de la mayoría de los sistemas de reconocimiento de imágenes modernos de la actualidad, que se basan en técnicas jerárquicas de aprendizaje profundo y han demostrado ser una solución poderosa cuando se aplican a una gran variedad de tareas de reconocimiento complejas.

Hoy en día, no cabe duda de que la CNN se ha convertido en el estándar para la visión artificial. Sin embargo, el problema es que estos sistemas son computacionalmente costosos y deben implementarse en ámbitos grandes de computación. Por lo tanto, es muy difícil adaptar los algoritmos necesarios para este tipo de tarea en un entorno móvil donde la capacidad computacional es limitada. El objetivo principal de este trabajo es demostrar cómo se puede implementar un sistema basado en CNN en entornos de baja energía como dispositivos móviles y embebidos, sin dejar de ser capaz de realizar la compleja tarea de detección de objetos en tiempo real.

Contribuciones

Las contribuciones específicas introducidas en este trabajo son las siguientes:

1. Un algoritmo para preprocesar imágenes y extraer características destacadas mientras se mantiene una distribución de datos normalizada que se adapta a los dispositivos con cámaras integradas.
2. Un marco de red neuronal para diseñar arquitecturas que requieren recursos computacionales mínimos para realizar reconocimiento de imágenes en entornos de baja energía.

3. Un algoritmo para compartir mapas de activación de la red base de un clasificador con el fin de extender la tarea de clasificación a una tarea de detección de objetos.
4. Un algoritmo para combinar los resultados de la inferencia sobre imágenes extensas para encontrar objetos íntegros.
5. Una metodología de inferencia basada en algoritmos conjuntos para proporcionar información de contexto y aumentar la robustez de las detecciones.
6. Una descripción de cómo implementar de manera más eficiente la red neuronal propuesta al enfocar mejor los recursos de hardware disponibles en dispositivos móviles y embebidos.

Organización

En este trabajo se profundiza en el desarrollo de un sistema de visión artificial basado en redes neuronales, con especial énfasis en su implementación en dispositivos de baja potencia.

En el Capítulo 2, se da una breve revisión de las redes neuronales y su aplicación a la visión artificial a través de redes neuronales convolucionales. Se describen los tipos de capas que se utilizan comúnmente para este tipo de red y se establece la conexión entre las CNN y los sistemas de visión por computadora de inspiración biológica.

El tema principal de esta tesis aparece en el Capítulo 3, donde se describe la arquitectura de propuesta de red neuronal, junto con los algoritmos correspondientes para llevar a cabo la tarea de detección de objetos en entornos de baja potencia. El capítulo comienza explicando el procesamiento de imágenes en los dispositivos móviles y describe cómo adaptar el algoritmo de normalización de contraste. A continuación, se proporciona la arquitectura básica de la red neuronal de clasificación, con una explicación sobre ciertos elementos de diseño que la hacen más aplicable a los dispositivos móviles. Luego, este capítulo explica cómo esta red troncal puede extenderse para manejar el problema de la localización adaptando la arquitectura de la red usando mapas convolucionales compartidos. Finalmente, se proporciona un algoritmo para analizar y colapsar múltiples resultados vecinos de detección en un mapa compartido, que es una mejor alternativa para los sistemas móviles.

A continuación, el Capítulo 4 ofrece una descripción general de las características del hardware y el entorno de los dispositivos móviles y embebidos. Se revisa la información sobre las unidades de cómputo que se encuentran en dicho hardware, junto con una discusión sobre cómo las rutas de acceso a la memoria pueden afectar la implementación de estos sistemas. A esto le sigue una explicación detallada de los diferentes métodos de optimización que se pueden utilizar para implementar el software que ejecuta el sistema propuesto.

El capítulo 5 proporciona la metodología de experimentación y da los resultados obtenidos. Se describe una aplicación de referencia de muestra y su modelo de red neuronal correspondiente. Esta aplicación se utiliza luego como base para realizar varios experimentos, que ayudan a establecer la exactitud de los métodos propuestos. El Capítulo 6 tiene un objetivo similar pero en un caso de aplicación más detallado. Este capítulo detalla el desarrollo de una aplicación de detección de orejas, explicando cómo el método propuesto permite la implementación de dicho sistema en un dispositivo de hardware portátil compacto.

El trabajo finaliza con el Capítulo 7 que aporta las conclusiones de este trabajo y plantea algunos puntos de discusión y futuras líneas de investigación.

Conclusiones

En este trabajo se presentó un sistema eficiente de detección de objetos basado en redes neuronales convolucionales profundas. Todas las decisiones de diseño involucradas en el desarrollo de este sistema fueron descritas en gran detalle, aunque hubo algunas sutilezas en los conceptos de implementación y programación que por brevedad no se mencionaron aquí.

Se describió un método para preprocesar datos de imágenes, que prepara la imagen a través del espacio de color YUV tal como la proporcionan los ISP en los sistemas móviles. El algoritmo aprovecha elementos de diseño como la compresión de la información de color y la disponibilidad de una operación convolucional optima que puede ejecutarse eficientemente en hardware. Se proporcionó la descripción de una arquitectura base de red neuronal, que utiliza una cantidad mínima de conexiones y neuronas para producir resultados de clasificación. Esta red se adaptó y amplió con un método llamado mapas compartidos para transformar la capacidad de clasificación a una capacidad de localización, lo cual es esencial para la detección de objetos. El último elemento de la arquitectura propuesta fue un algoritmo de

inferencia discreta que utiliza la minimización de energía como método para reducir las detecciones activas y encontrar la ubicación final de un objeto.

También se proporcionó una descripción de cómo implementar de manera eficiente este sistema de red neuronal, dadas las unidades de cómputo de hardware comúnmente disponibles en los dispositivos embebidos. Se repasaron las características de hardware y de diferentes elementos de cómputo, y una descripción de cómo se pueden programar con software teniendo en mente la optimización específica para sistemas de redes neuronales.

Esta tesis ha demostrado que es posible desarrollar un sistema de detección de objetos basado en CNN altamente eficiente que sea capaz de funcionar en entornos restringidos. En comparación con los sistemas de detección de estado del arte, esta arquitectura proporciona una precisión limitada, pero con un tiempo de ejecución mucho más rápido, una medida que se traduce a la cantidad de energía gastada en dispositivos de computación embebidos.

El trabajo que aquí se presenta abre muchas posibilidades interesantes con respecto a los tipos de sistemas de visión artificial que se pueden implementar en estos dispositivos. Si bien esta arquitectura puede no ser lo suficientemente poderosa para impulsar sistemas críticos de toma de decisiones, puede proporcionar una solución para la detección masiva en implementaciones de dispositivos a gran escala. También proporciona un marco sobre el cual seguir construyendo y seguir desarrollando sistemas cada vez más capaces.

A medida que la tecnología continúa avanzando y los dispositivos embebidos continúan volviéndose más económicos, es natural el continuar ampliando el tamaño y la complejidad de los sistemas que se pueden implementar en ellos. Sin embargo, muchas de las consideraciones de optimización establecidas en este trabajo continuarán teniendo un impacto en las generaciones futuras de de visión artificial móvil.

Como futuras líneas de investigación sería de gran interés estudiar la implementación eficiente de otro tipo de sistemas de redes neuronales, por ejemplo, redes neuronales recurrentes capaces de analizar secuencias de imágenes para tareas como la predicción de comportamiento y acciones.

Para concluir, este ha demostrado ser un proyecto de investigación muy interesante, pero esto es solo el comienzo de lo que seguramente será un campo fructífero en el cual seguir trabajando.

Bibliography

[1] S. Lazebnik, C. Schmid, J. Ponce, *CVPR 06: Proceedings of the 2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition* **2006**, *2*, 2169–2178.

[2] H. Bay, A. Ess, T. Tuytelaars, L. V. Gool, *Computer Vision and Image Understanding* **2008**, *110*, Similarity Matching in Computer Vision and Multimedia, 346 –359, DOI 10.1016/j.cviu.2007.09.014.

[3] N. Dalal, B. Triggs in Computer Vision and Pattern Recognition, 2005. CVPR 2005. IEEE Computer Society Conference on, *Vol. 1*, **2005**, 886–893 vol. 1, DOI 10.1109/CVPR.2005.177.

[4] K. Jarrett, K. Kavukcuoglu, M. Ranzato, Y. LeCun in 12th International Conference on Computer Vision, IEEE, **2009**, pp. 2146–2153.

[5] D. H. Hubel, T. N. Wiesel, *The Journal of Physiology* **1959**, 574–591.

[6] K. Fukushima, *Biological Cybernetics* **1980**, *36*, 93–202.

[7] Y. LeCun, L. Bottou, Y. Bengio, P. Haffner in Proceedings of the IEEE, IEEE, **1998**.

[8] A. Krizhevsky, I. Sutskever, G. Hinton in *Advances in Neural Information Processing Systems*, (Eds.: P. Bartlett, F. Pereira, C. Burges, L. Bottou, K. Weinberger), 25, **2012**, pp. 1106–1114.

[9] D. C. Ciresan, U. Meier, L. M. Gambardella, J. Schmidhuber in 11th International Conference on Document Analysis and Recognition, ICDAR, **2011**.

[10] D. C. Ciresan, U. Meier, J. Masci, J. Schmidhuber, *Neural Networks* **2012**, 333–338.

[11] A. Newell, H. A. Simon, *Commun. ACM* **Mar. 1976**, *19*, 113–126, DOI 10.1145/360018.360022.

[12] P. Wang in, *Vol. 8*, **Jan. 2007**, pp. 31–62, DOI 10.1007/978-3-540-68677-4_2.

[13] D. C. Ciresan, U. Meier, J. Masci, L. M. Gambardella, J. Schmidhuber in Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence, **2011**, pp. 1237–1242.

[14] Y. LeCun, L. Bottou, Y. Bengio, P. Haffner in Proceedings of the IEEE, **1998**.

[15] A. Krizhevsky, I. Sutskever, G. E. Hinton in, (Eds.: P. Bartlett, F. Pereira, C. Burges, L. Bottou, K. Weinberger), 25, **2012**, pp. 1106–1114.

[16] D. Ciresan, U. Meier, L. Gambardella, J. Schmidhuber, *In: 11th International Conference on Document Analysis and Recognition. ICDAR.* **2012**.

[17] T. Wang, D. J. Wu, A. Coates, A. Y. Ng in ICPR, IEEE, pp. 3304–3308.

[18] S. Ji, W. Xu, M. Yang, K. Yu in IEEE Transactions on Pattern Analysis and Machine Intelligence, *Vol. 35*, IEEE, **2013**, pp. 221–231.

[19] S. Lawrence, C. Giles, A. C. Tsoi, A. Back, *Neural Networks IEEE Transactions on* **1997**, *8*, 98–113.

[20] D. Ciresan, U. Meier, J. Masci, J. Schmidhuber, *Neural Networks.* **2012**, *32*, 333–338.

[21] J. H. Byrne, Neuroscience Online, http://neuroscience.uth.tmc.edu/, University of Texas Medical School, **1997**.

[22] D. E. Rumelhart, G. E. Hinton, R. J. Williams, *Nature* **1986**, *323*, 533–536.

[23] A. Ng, Unsupervised Feature Learning and Deep Learning Tutorial, Computer Science Department, Stanford University, **2011**.

[24] Y. LeCun, Y. Bengio, Convolutional Networks for Images, Speech, and Time-Series, tech. rep., AT&T Bell Laboratories, **1995**.

[25] H. Lee, Y. Largman, P. Pham, A. Ng in *Advances in Neural Information Processing Systems*, 22, **2009**, pp. 1096–1104.

[26] S. Ji, W. Xu, M. Yang, K. Yu in IEEE Transactions on Pattern Analysis and Machine Intelligence, *Vol. 35*, IEEE, **2013**, pp. 221–231.

[27] K. Suzuki, *International Journal of Biomedical Imaging* **2012**.

[28] R. C. Gonzalez, R. E. Woods, *Digital Image Processing*, 3rd Edition, Prentice Hall, **2007**.

[29] T. Serre, L. Wolf, T. Poggio, **2005**, 994–1000.

[30] M. Riesenhuber, T. Poggio, *Nature Neuroscience* **1999**, *12*, 1019–1025.

[31] R. Girshick, J. Donahue, T. Darrell, J. Malik, Rich feature hierarchies for accurate object detection and semantic segmentation, **2014**.

[32] R. Girshick, Fast R-CNN, **2015**.

[33] S. Ren, K. He, R. Girshick, J. Sun, Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks, **2016**.

[34] K. He, G. Gkioxari, P. Dollár, R. Girshick, Mask R-CNN, **2018**.

[35] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, A. C. Berg, *Lecture Notes in Computer Science* **2016**, 21–37, DOI 10.1007/978-3-319-46448-0_2.

[36] J. Redmon, S. Divvala, R. Girshick, A. Farhadi, You Only Look Once: Unified, Real-Time Object Detection, **2016**.

[37] J. Redmon, A. Farhadi, YOLO9000: Better, Faster, Stronger, **2016**.

[38] J. Redmon, A. Farhadi, YOLOv3: An Incremental Improvement, **2018**.

[39] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, H. Adam, MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications, **2017**.

[40] K. He, X. Zhang, S. Ren, J. Sun, Deep Residual Learning for Image Recognition, **2015**.

[41] T.-Y. Lin, M. Maire, S. Belongie, L. Bourdev, R. Girshick, J. Hays, P. Perona, D. Ramanan, C. L. Zitnick, P. Dollár, Microsoft COCO: Common Objects in Context, **2015**.

[42] M. Livingstone, *Vision and Art: The Biology of Seeing*, Abrams, **2008**, pp. 12–83.

[43] N. Pinto, D. D. Cox, J. J. Dicarlo, *PLoS Computational Biology* **2008**, *4*.

[44] S. Lyu, E. Simoncelli in Conference on Computer Vision and Pattern Recognition, IEEE, **2008**, pp. 1–8.

[45] S. Treitel, J. L. Shanks, *IEEE Transactions on Geoscience Electronics* **1971**, *9*, 10–27, DOI 10.1109/TGE.1971.271457.

[46] A. Jordao, R. Kloss, F. Yamada, W. R. Schwartz, Pruning Deep Neural Networks using Partial Least Squares, **2019**.

[47] E. J. Crowley, J. Turner, A. Storkey, M. O'Boyle, A Closer Look at Structured Pruning for Neural Network Compression, **2019**.

[48] J Pearl, *Artif. Intell.* **Sept. 1986**, *29*, 241–288, DOI 10.1016/0004-3702(86)90072-X.

[49] P. F. Felzenszwalb, D. P. Huttenlocher, *International Journal of Computer Vision* **2006**, *70*, 41–54.

[50] A. Rosenfeld, M. Thurston, *IEEE Transactions on Computers* **1971**, *C-20*, 562–569, DOI 10.1109/T-C.1971.223290.

[51] N. Bodla, B. Singh, R. Chellappa, L. S. Davis, Soft-NMS – Improving Object Detection With One Line of Code, **2017**.

[52] G. Wang, J. Wu, *Guide to Three Dimensional Structure and Motion Factorization*, Springer, **2011**, pp. 29–41.

[53] L. Bottou in *Neural Networks, Tricks of the Trade, Reloaded*, Lecture Notes in Computer Science (LNCS), Springer, **2012**, pp. 430–445.

[54] M. Everingham, L. V. Gool, C. K. I. Williams, J. Winn, A. Zisserman, *International Journal of Computer Vision* **2009**, *88*, Printed version publication date: June 2010, 303–308.

[55] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, L.-C. Chen, MobileNetV2: Inverted Residuals and Linear Bottlenecks, **2019**.

[56] P. Galdámez, M. González Arrieta, R. R. M., *17th International Conference on Information Fusion (FUSION)* **2014**, 1–6.

[57] M. Burge, W. Burger, *ICPR* **2000**.

[58] D. Hurley, M. Nixon, J. Carter., *Image and Vision Computing* **2002**.

[59] P. Yan, K. W. Bowyer, *In Proceedings of International Conference on Computer Vision and Pattern Recognition-Workshop volume 3* **2005**.

[60] E. Gonzalez., *http://www.ctim.es/research_works/ami_ear_database/* **2015**.

[61] P. J. Flynn, K. W. Bowyer, P. J. Phillips, *Audioand Video-Based Biometric Person Authentication* **2003**.

[62] K. Chang, K. W. Bowyer, S. Sarkar, B. Victor, *IEEE Transactions on Pattern Analysis and Machine Intelligence* **2003**.

[63] R. Raposo, E. Hoyle, A. Peixinho, H. Proença., *IEEE Workshop on Computational Intelligence in Biometrics and Identity Management* **2011**.

[64] G. Bradski, *Dr. Dobb's Journal of Software Tools.* **2000**.