

HBONEXT: AN EFFICIENT DNN FOR LIGHT EDGE EMBEDDED DEVICES

by

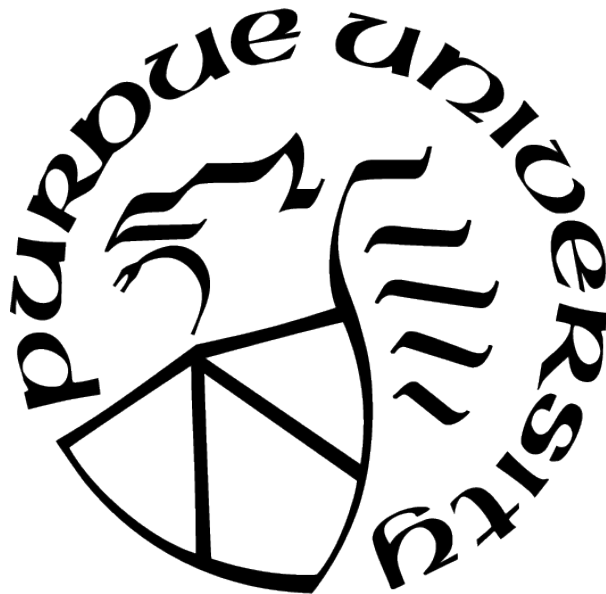
Sanket Ramesh Joshi

A Thesis

Submitted to the Faculty of Purdue University

In Partial Fulfillment of the Requirements for the degree of

Master of Science



Department of Electrical and Computer Engineering

Indianapolis, Indiana

May 2021

**THE PURDUE UNIVERSITY GRADUATE SCHOOL
STATEMENT OF COMMITTEE APPROVAL**

Dr. Mohamed El-Sharkawy, Chair

Department of Electrical and Computer Engineering at IUPUI

Dr. Brian King

Department of Electrical and Computer Engineering at IUPUI

Dr. Maher Rizkalla

Department of Electrical and Computer Engineering at IUPUI

Approved by:

Dr. Brian King

Dedicated to my grandparents

ACKNOWLEDGMENTS

This work was not possible without the constant support and words of wisdom received from Dr. El-Mohamed El-Sharkawy. He has been a constant source that kept me motivated to pursue this research work.

Dr. Brian King and Dr. Maher Rizkalla for being on my committee and giving timely reviews that culminated in major results in my study.

I would like to thank the entire department of Electrical and Computer Engineering at IUPUI, especially Sherrie Tucker, for her constant helpful advice and eagerness to assist and direct.

To my mentor Li-Cheng Tsung for providing minute insights on how to investigate and troubleshoot.

I would like to express my gratitude to my colleagues at the IoT Collaboratory Lab, especially Jayan Kant Duggal and Maneesh Ayi, for ensuring an excellent working environment.

My parents, Manik and Ramesh Narayan Joshi, as well as my sisters, Rashmi Sagar Joshi and Vaishali Samir Joshi, deserve a special thank you; without them, not a single step would have been possible to ascend.

Finally, I would like to thank all my friends for always making me feel at ease.

TABLE OF CONTENTS

LIST OF TABLES	7
LIST OF FIGURES	8
ABBREVIATIONS	10
ABSTRACT	12
1 INTRODUCTION	14
1.1 Challenges	15
1.2 Motivation	16
1.3 Research Objective	17
2 LITERATURE REVIEW	18
2.1 Early Development	18
2.2 Deep Architectures: Evolution	20
2.3 Convolutional Neural Network: An Overview	22
2.3.1 Input Image	25
2.3.2 Convolutional Layers	26
2.3.3 Pooling Layers	26
2.3.4 Activation Layers	27
2.3.5 Fully Connected Layers	27
2.4 Gradient Descent	27
2.5 Parameter Calculations	28
2.6 Training Process of CNNs	30
2.7 Visualization	31
2.8 Measure of Performance	31
2.9 Related Architectures	32
2.9.1 HBONet: Harmonious Bottleneck Network (Baseline)	32
2.9.2 Flipped Inverted Residual (FIR): Sandglass Structure	34
3 HARDWARE AND SOFTWARE	36
3.1 Hardware Used	36
3.1.1 NXP Bluebox 2.0 (BLBX2)	37

3.1.2	Features and Specifications of BLBX2	39
3.1.3	Vision Processor – S32V234	41
3.1.4	Processor - LS2084A	42
3.2	Software Used	43
3.2.1	Real-Time Multi Sensor Application (RTMaps)	43
3.2.2	Various Deep Learning Frameworks	44
4	DESIGN EXPLORATION AND MODIFICATIONS	46
4.1	Techniques to Improve Performance	46
4.2	Use of Optimizer	46
4.3	Learning Rate Scheduling Methods	48
4.4	Use of Activation Function	49
4.5	Checkpoint Save/Load Method	50
5	PROPOSED HBONEXT ARCHITECTURE	51
5.1	Design Considerations	52
5.2	Width Multiplier Consideration	54
6	RESULTS AND DISCUSSIONS	55
6.1	Model Accuracy	55
6.2	Reducing Overfitting	56
6.2.1	Data Augmentation	56
6.2.2	Dropout	57
6.3	Model Size Variants	59
6.4	Hardware Validation	60
6.4.1	NXPs BLBX2 Implementation Steps	60
6.4.2	NXPs BLBX2 Implementation Results	61
7	CONCLUSIONS AND FUTURE WORK	66
7.1	Conclusion	66
7.2	Future Work	67
	REFERENCES	69
	PUBLICATIONS	72

LIST OF TABLES

2.1	CNN Parameter Calculations	29
2.2	HBONet Architecture	34
5.1	HBONext v1.1 Architecture	51
5.2	HBONext v1.2 Architecture	53
6.1	Comparison: Total Number of Parameters	56
6.2	Performance Comparison for HBONext for $w=1.0$	58
6.3	Width Multiplier Variants of HBONext v1.1	60
6.4	Key Implementation Parameters of HBONext v1.2	62

LIST OF FIGURES

2.1	Timeline: AI Advancement	18
2.2	Venn Diagram Representation	19
2.3	Evolution of ANNs	21
2.4	General Architecture of CNNs	23
2.5	Example of Convolution Operation $k = 3 \times 3, s = 1$	23
2.6	Example of Convolution Operation with Zero Padding	24
2.7	Network Input: RGB Image	25
2.8	Pooling Methods	26
2.9	Theory Behind a Local Minimum	27
2.10	Illustration: Calculations of the Number of Parameters	28
2.11	CNN Visualization (Input: School bus)	31
2.12	Operational Bottleneck Block of HBONet	32
2.13	Inverted Residual with Linear Bottleneck and Harmonious Bottleneck, with strides	33
2.14	Residual blocks (a) Traditional Bottleneck Arrangement, (b) Block Sandglass Bottleneck Arrangement	35
3.1	BlueBox 2.0 by NXP	37
3.2	Levels of Autonomy	38
3.3	General Architecture of BLBX2	40
3.4	S32V234 Block Diagram	41
3.5	LS-2084A Block Diagram	42
3.6	RTMaps Remote Studio connected to BLBX2	43
3.7	DLL Frameworks	44
4.1	Comparison Between Different Optimizers	47
4.2	Performance of different L_r Schedules	48
4.3	ELU non-linear Activation Function	49
4.4	Activation Functions for Mish, ReLU, SoftPlus, and Swish	50
5.1	Harmonious Bottleneck Design with Different Strides	52

6.1	Accuracy vs the Number of Epochs: (a) HBONet (Baseline), (b) HBONext v1.1 (Proposed Architecture)	55
6.2	CIFAR-10 Dataset (a) without Augmentation, (b) with Augmentation	57
6.3	HBONext v1.2 (a) Accuracy and Losses vs Epochs (Augmentation)	58
6.4	HBONext v1.2 (b) Accuracy and Losses vs Epochs (Augmentation + L_r Scheduling)	59
6.5	Interfacing Overview with NXP BLBX2	60
6.6	HBONext Architecture Testing on RTMaps	62
6.7	Image Classifier using HBONext v1.2 on RTMaps	63
6.8	Image Classification for CIFAR-10 using HBONext v1.2	63
6.9	RTMaps Console Output	64
6.10	TeraTerm Console Validation Results	65
7.1	Example for Object Detection	67

ABBREVIATIONS

ADAS	Advanced Driver Assistant System
AI	Artificial Intelligence
BLBX2	BlueBox 2.0
CNNs	Convolutional Neural Networks
CMYK	Cyan, Magenta, Yellow, Black
CIFAR10	Canadian Institute for Advanced Research
CPUs	Central Processing Units
DCNNs	Deep Convolutional Neural Network
DHbneck	Derived Harmonious Bottlenecks
DL	Deep Learning
DWConv	Depth-Wise Convolution
EltAdd	Element-by-Element Addition
ELU	Exponential Linear Unit
FC	Fully Connected
FIR	Flipped Inverted Residual
FPGA	Field Programmable Gate Array
GD	Gradient Descent
GPUs	Graphic Processing Units
GPS	Global Positioning System
GTX	Giga Texel Shader eXtreme
HBO	Harmonious Bottlenecks on two Orthogonal dimensions
L_r	Learning Rate
LiDAR	Light Detection and Ranging
MB	Mega Byte
ML	Machine Learning
MNIST	Modified National Institute of Standards and Technology
NN	Neural Net
PC	Personal Computer

PWConv	Point-Wise Convolution
RGB	Red, Green, Blue
RST	Reset
RTMaps	Real-Time Multisensor Applications
ROS	Robotics Operating System
SD	Secure Digital
SDK	Software Development Kit
SGD	Stochastic Gradient Descent
SSD	Single Shot Detector
SVM	Support Vector Machines
TCP/IP	Transmission Control and Internet Protocol
TPUs	Tensor Processing Units
UART	Universal Asynchronous Receiver-Transmitter
USB	Universal Serial Bus

ABSTRACT

Every year the most effective Deep learning models, CNN architectures are showcased based on their compatibility and performance on the embedded edge hardware, especially for applications like image classification. These deep learning models necessitate a significant amount of computation and memory, so they can only be used on high-performance computing systems like CPUs or GPUs. However, they often struggle to fulfill portable specifications due to resource, energy, and real-time constraints. Hardware accelerators have recently been designed to provide the computational resources that AI and machine learning tools need. These edge accelerators have high-performance hardware which helps maintain the precision needed to accomplish this mission.

Furthermore, this classification dilemma that investigates channel interdependencies using either depth-wise or group-wise convolutional features, has benefited from the inclusion of Bottleneck modules. Because of its increasing use in portable applications, the classic inverted residual block, a well-known architecture technique, has gotten more recognition. This work takes it a step forward by introducing a design method for porting CNNs to low-resource embedded systems, essentially bridging the difference between deep learning models and embedded edge systems. To achieve these goals, we use closer computing strategies to reduce the computer's computational load and memory usage while retaining excellent deployment efficiency. This thesis work introduces HBONext, a mutated version of Harmonious Bottlenecks (DHbneck) combined with a Flipped version of Inverted Residual (FIR), which outperforms the current HBONet architecture in terms of accuracy and model size miniaturization. Unlike the current definition of inverted residual, this FIR block performs identity mapping and spatial transformation at its higher dimensions. The HBO solution, on the other hand, focuses on two orthogonal dimensions: spatial (H/W) contraction-expansion and later channel (C) expansion-contraction, which are both organized in a bilaterally symmetric manner. HBONext is one of those versions that was designed specifically for embedded and mobile applications. In this research work, we also show how to use NXP Bluebox 2.0 to build a real-time HBONext image classifier. The integration of the model into this hardware has been a big hit owing to the limited model size of 3 MB. The model was trained and vali-

dated using CIFAR10 dataset, which performed exceptionally well due to its smaller size and higher accuracy. The validation accuracy of the baseline HBONet architecture is 80.97%, and the model is 22 MB in size. The proposed architecture HBONext variants, on the other hand, gave a higher validation accuracy of 89.70% and a model size of 3.00 MB measured using the number of parameters. The performance metrics of HBONext architecture and its various variants are compared in the following chapters.

1. INTRODUCTION

The origins of Convolutional neural networks (CNNs) can be traced back to both neuroscience and artificial intelligence. They were influenced by early discoveries in the field of vision-based biological research. Artificial neural networks are based on the neurons discovered inside the human brain, and linked together using computer science, mathematics, and engineering fundamentals. CNNs have been used for visual purposes since the early 1980s, but their development slackened in the mid-2000s. However, in 2012, the popularity of larger labeled datasets, improved algorithmic techniques, and the need for more computing power propelled it to the forefront of a neural network renaissance, which has accelerated since then [1]. Deep Learning models, on the other hand, have transformed artificial intelligence prospects by solving a variety of key problems such as identification, regression, and many others. DL models are more complex versions of ANNs, consisting of a larger number of layers connected by various weights to learn data features with multiple level of abstraction. They have assisted in the construction of automated machines that can perform activities in a variety of fields, including medicine, self-driving vehicles, image processing, and data science, that are comparable to or even better than humans [2].

ANNs with many hidden layers have been widely investigated for their robust learning ability, which can be improved by increasing the number of hidden layers in a short period of time by adding depth. As a result, “deep” learning refers to a type of machine learning improvement that can handle complicated shapes and artifacts in huge datasets. The basics, pros, and cons of Machine Learning for deep architectures are thoroughly discussed by Goodfellow [3]. The development of various datasets and virtual simulators, as well as recent developments in deep learning architectures and computer vision [4], [5], have made it clear that widespread efforts are being made in the autonomous driving research field, which is blooming with every passing day[6]. Since most car accidents today are caused by human negligence, the introduction of self-driving cars can minimize traffic congestion, fuel usage and accidents, and save human lives.

The collaborative European project known as ‘PROMETHEUS’ [7], [8] took the first steps in the autonomous driving realm, decomposing the task of driving into smaller chunks

from perception to strategy, which helped in processing. ImageNet classification challenge that started in 2012 has been a driving force to develop today's best performing CNN architectures, which are growing deeper and becoming more efficient with every passing year [9]. With the increasing quest of improving these architectures for higher accuracy yield, many sophisticated mathematical approaches are under development but at the cost of higher computational and storage requirements.

Many novel studies have emerged to build further light-weight CNN models that are compatible and feasible for actual real-time implementation and to investigate this critical problem. With all the research in mind, the goal of this thesis is to learn about the evolution of deep CNN (DCNN) technology and its modular architectures, particularly for image classification and its validation on light embedded edge devices. It employs methods from its forerunners all the way up to the most cutting-edge mathematical methods in the deep learning realm. It also provides some interesting directions in a short manner. It also briefly discusses some promising techniques for further architectural modifications to achieve better performance.

1.1 Challenges

Despite deep CNN's (DCNN) good image classification efforts over the years, there are still barriers that need to be discussed. Implementation on the embedded edge platforms for mobile development, upgrading contemporary datasets, understanding deep and multilabel imagery, and model training methods are just a few of the obstacles. The most recent DCNNs rely on supervised training strategies and make probabilistic projections based on a specific dataset for the sole purpose of experimentation. As a result, they are unable to take advantage of the vast amounts of unlabeled data that are publicly accessible. These deep networks often rely heavily on hyperparameters such as the number of training epochs, learning rate, optimizer, activation function, learning rate schedulers, and so on. However, calculating these values for an algorithm depend on expertise in the field, some thumb rules, or any established parameter search techniques that has previously researched and helped. Another strategy for mobile development of these deep models is to factorize the

convolutional filters from $N \times N$ to $1 \times N$, followed by $N \times 1$ convolutions, which has resulted in a significant reduction in model size. This benefit is being utilized in this work, which makes widespread use of depth-wise convolution followed by point-wise convolutions. The most important thing is that every effort is made to close the theoretical gap between deep CNNs and biological neural networks as much as possible. This article [10], focuses on how the brain performs priority assignments in a manner analogous to what a back propagation does in our machine algorithms, and it is viewed as the most crucial step in linking our deep neural approaches to human brain neuron operations.

1.2 Motivation

The abrupt increase in the number of cars on the road has made us realize the importance of safety, connectivity, attentive contributions to prevent collisions, and a well-managed transportation system. The Intelligent Transportation has now recognized as one of the most effective ways to navigate us through increasing traffic flow, with safety, and end-user control. Collision avoidance, lane keeping, emergency braking, and other intelligent techniques have already been adopted by the automobile industry [11]. All the auto firms are now operating fleets to collect as much data as possible from the environment to better train these imagery algorithms, which are expected to revolutionize the transportation industry. Investing in new road infrastructure is not a viable solution but using transportation data gathered from various sensors such as the Global Positioning System (GPS), ultrasonic sensors, LiDAR, and on-board cameras is the only way to improve transportation efficiency, safety, and security. Deep CNNs (DCNNs) have received a lot of attention in the research community as a credible option for this computer vision task of making judgments about the surrounding objects for a vehicle in motion. Today's smart autonomous cars include these models in their advanced driver assistance toolkit, which assist users with traffic sign recognition, collision warnings, lane keeping, and pedestrian detection, among other things.

Our focus has switched to a machine making decisions for our safety and smooth travel because of this new field of artificial intelligence. This is akin to a newborn infant attempting to comprehend and learn about the world around them solely using a projected image in front

of them. This research work seeks to make a small contribution to the field of deep CNN by using various mathematical modification methods, or in other words, to investigate the design space. Even though we intend to use strong GPU resources to train the architecture using a supervised procedure using available labeled data, the performance of these models on the embedded edge is of paramount importance in terms of maintaining precision while also maintaining the achieved speed and model size.

1.3 Research Objective

The aim of this research is to delve into the field of deep CNNs and learn about the different techniques that can be used to help the architecture outperform its baseline features. Here are few points that have been thoroughly explored to better understand this design modification space.

- Understanding the baseline HBONet model architecture under consideration.
- Obtaining the HBONet baseline architecture results using CIFAR-10 dataset.
- Modifying the baseline HBONet architecture.
- Obtaining the modified HBONext v1.1 result using CIFAR-10 dataset.
- Comparison of the achieved results and visualization in real-time.
- The impact of Parameter Tunning.
- Use of different optimizer and learning rate scheduling techniques.
- Use of data augmentation technique.
- Proposing HBONext v1.2 (a), (b) and its validation.
- Hardware deployment of this modified DCNN architecture and Validation.

2. LITERATURE REVIEW

In neuroscience, computational models are used for a variety of purposes. They can verify intuitions about how a system works by allowing those intuitions to be directly tested. They also provide a way to test new theories in an optimal laboratory setting, where every detail can be monitored and controlled. Models also open a new realm of understanding for the system in question. Models often use parametric model to open a new realm of interpretation for the system in question. Convolutional neural networks (CNNs) have lately been used as a model of the visual field to perform all these operations. This segment covers the original growth and progress in CNNs from its predecessors to the successful deep CNN revival.

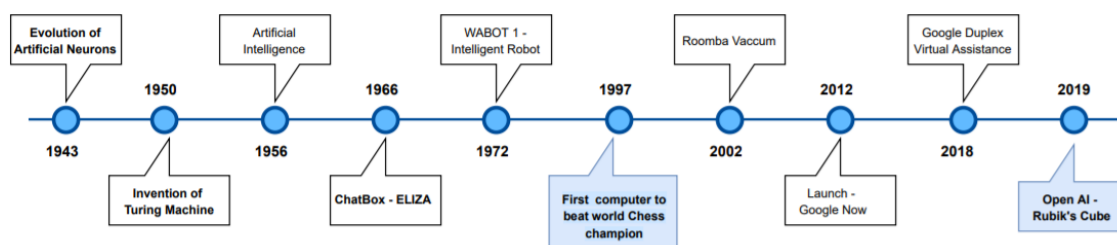


Figure 2.1. Timeline: AI Advancement

The advancement in artificial intelligence and the development of this technology with each passing year has resulted in the discovery of smart intelligent machines with human-like decision-making abilities, or perhaps greater than humans.

2.1 Early Development

The term “artificial neurons” was coined in 1943 by neuro physicist Warren McCulloch and mathematician Walter Pitts, who presented their findings on neurons in the human brain. They build on their study by incorporating a basic Neural Net (NN) from an electrical circuit perspective[12]. The history started to build in early 1959, with Wiesel and Hubel supervising neurobiological research [13], [14]. They found that neurons at various levels of the visual system behaved differently with different stimuli patterns and replied strongly to specifically directed light patterns, like bars, but overlooked more complicated patterns of the input signal, which resulted into intense responses from neurons later. With the

multilayered architecture ‘neocognitron’ implemented in 1979, the network was effective in recognizing different input patterns regardless of any change or distortion in form [15]. This laid the foundation for today’s CNN family. Werbos [16] and Rumelhart [17] introduced backpropagation for the first time in 1981 and 1986, respectively.

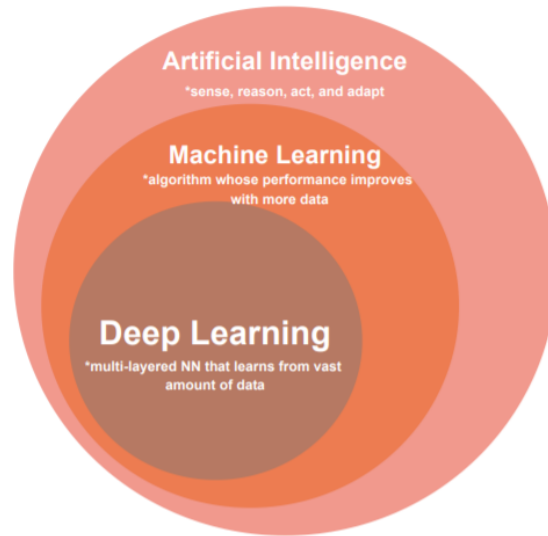


Figure 2.2. Venn Diagram Representation

This shows how this algorithm can be used to train an ANN’s internal hidden neuron to represent important features. In 1989, LeCun introduced the first multilayered CNNs, which were used to classify real-world images such as handwritten digits and codes [18]. These models were trained using backpropagation in a supervised manner, allowing a more automated learning process rather than hard coding for feature extraction unlike the predecessor techniques. Later, LeCun developed the original CNN (LeNet-5) [19] with the goal of using it in a general textual recognition application for individual character identification. This led to the publication of the MNIST dataset, which contains approximately 70,000 handwritten digits, as a useful dataset for any computer vision task involving image classification. As a result, deep CNNs (DCNN) began to emerge and conquer. Between 1990 and early 2000, NN research came to a halt due to common conviction that gradient descent would never recover from the low local minima problem. Other statistical techniques, such as the Scalar Vector Machine (SVM) [20], began to appear during this era, and some CNNs

based on these principles were proposed. This research was used by Microsoft Office Toolkit for handwritten digit/character recognition in English, Arabic, and East Asian languages, which are known to be early image classification applications of the time.

In 1965, Ivakhnenko and Lapa were the first to use feedforward multilayered NN, and their methods were the first ever deep CNN (DCNN) systems. This work fueled the unsupervised training method, in which sparse features that were locally invariant and warped were used to train DCNN-like architectures in an unsupervised fashion. This work also advocated the use of max pooling, which yielded excellent findings that were closer to available metrics such as MNIST and CALTECH-101. Weston [21], inspired a semi-supervised deep learning technique, but it received little recognition because it predicted errors with the MNIST dataset at the time. Unsupervised DCNN pretraining and semi-supervised embedding methods, on the other hand, regained interest. It began by performing unsupervised pseudo tasks on data and then transferring learnings to DCNNs. Backpropagation was used to train all DCNN layers, including classification layers. The final findings showed that information transfer accompanied by a controlled teaching phase aided in producing improved results and enhancing DCNN performance. The deep belief network pioneered unsupervised along with semi-supervised, and pretraining followed by supervised refining techniques. Deep Learning techniques are the subset of Machine learning, which turn is a subject of Artificial Intelligence.

2.2 Deep Architectures: Evolution

Over the span of a year, ANN has developed into more successful mathematical methods, allowing computers to imitate human behavior. The first generation of ANNs was a basic neural layer for Perceptron, and as we develop, so do these algorithms. Second generation used backpropagation to change neuron weights based on the error rate since they were constrained in simple computations. Then came Support Vector Machine algorithms, and then came Restricted Boltzmann Machine algorithms to solve the limitations of backpropagation, which made learning easier. To summarize Deep learning models have had a big impact in Supervised Learning, Unsupervised Learning, Reinforcement Learning, and Hybrid Learning

so far. When data is labelled, a classifier produces a probabilistic value for each class, which is referred to as supervised learning.

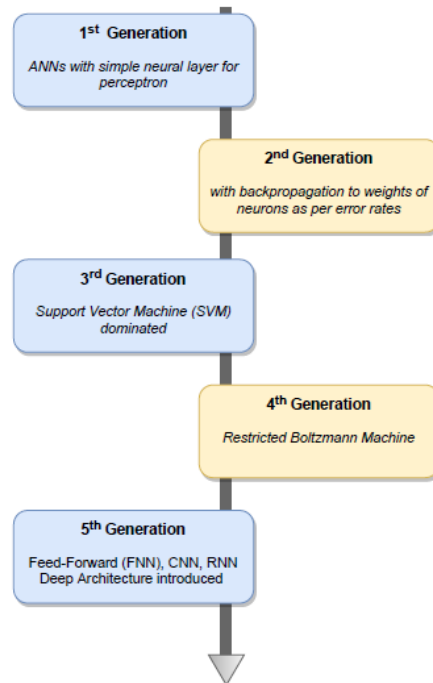


Figure 2.3. Evolution of ANNs

Unsupervised learning is applied on the known data and it later classifies it. Reinforcement Learning, on the other hand, is based on a reward and punishment system generated by a learning model and is widely used in games or for robots [22]. The research outbreaks as we transition from one generation to the next is seen in Figure 2.3, where deep CNNs seem to be crushing the classification problem. The field of deep learning has achieved following successes so far:

- Close to human level image classification.
- Close to human level speech recognition.
- Close to human level autonomous driving experience.
- Close to human level handwriting transcription.
- Capability to answer natural language answers.

- Google assistance, Alexa, Siri like technology
- Text to speech conversions
- Improved Web-based search
- ad-targeting used by Google, Bing, and Baidu

2.3 Convolutional Neural Network: An Overview

CNN is one of the most used deep learning networks[11]. The feature extraction phase and the classifier phase are the two most critical phases in these CNNs. The feature extraction process varies by model type, but it generally consists of a convolutional layer with finite filters that help convolve the input image and collect important features. Following it is the pooling layer, which supports flattened vector components and performs classification by reducing dimensionality or downsampling the input image to minimize processing and operations, and finally the fully connected layer, which helps flattened vector components and performs classification. Repeating the convolution-pool sequencing to retain further detail from the provided input data will further deepen the method. Convolutional layers take an input image, convolve it using various size filters, and perform matrix multiplications to reduce the number of weights and alter the model's variance. These filters can help detect edge information at a bottom level, and at a good extent, they can detect more objects and complex forms, which is a critical task for further image detection. Thanks to today's developments in GPUs or parallel computing type of training, CNNs have developed themselves as a key research area for growth in self-driving or any vision recognition industry.

Pooling layers, on the other hand, limit the number of neurons in a network, resulting in a smaller layer that captures more important features over sliding windows of input image data using a fixed length. To carry out this procedure, the most popular strategies are max pooling, average pooling, and adaptive pooling, which slide a set window over the layer and pick one raw value based on the type of pooling used. Max pooling chooses the largest of all and down samples weights to minimize computations and overfitting risks, while average pooling chooses the average of all the units.

The different layer learning parameters are bias, weight, kernel, stride, padding, and output dimension. **Bias** and **Weight** value are randomly allocated at first and modified as fed with data to lower the loss function. The difference between expected and predicted outcomes is represented by bias, while weight is the strength of the relation and determines its influence on the output. We incorporate a 2D area called **receptive field** (3×3) on this three-channel image dimension, which is nothing more than a **kernel** for extracting smaller, highly localized, and complex features.

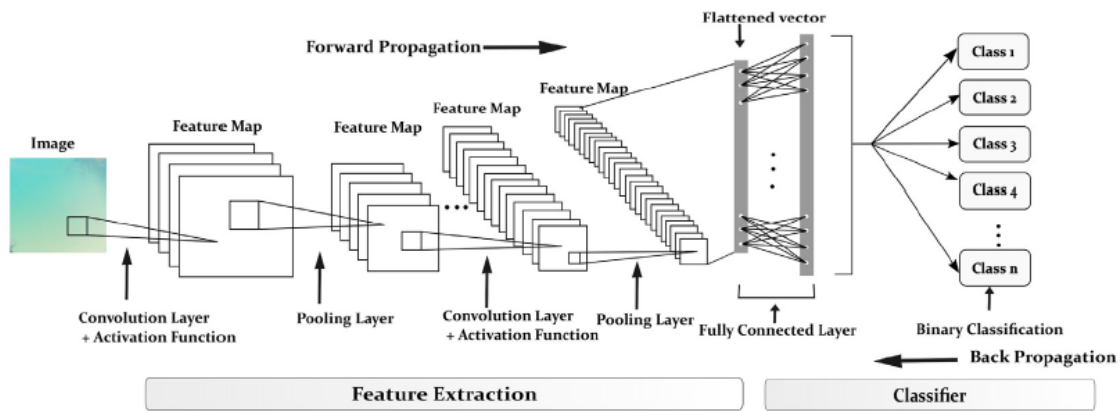


Figure 2.4. General Architecture of CNNs

The network gets stronger, weight sharing improves, and a greater volume of data is extracted as these filter sizes are used.

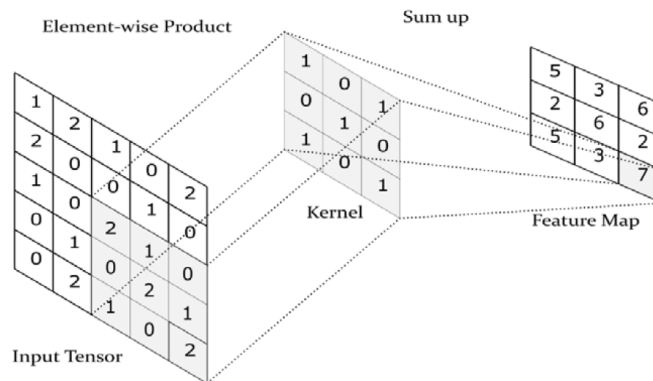


Figure 2.5. Example of Convolution Operation $k = 3 \times 3$, $s = 1$

Further research focuses on depth-wise separable convolution, which has proven to be one of the most effective strategies for making this feature extraction process more computationally powerful. Stride is the hyperparameter that governs the activity of the kernel around the input image. It is the distance between two consecutive kernel locations. As seen in Figure 2.5, it is an example of kernel 3x3 with a stride value of 1, in which we shift the filter one pixel at a time to create a convolved feature map. **Zero padding** is applied to address the issue of the size diminishing as the number of convolution layers grows.

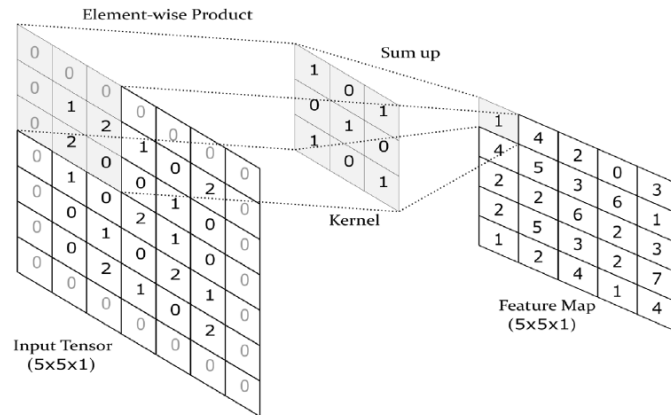


Figure 2.6. Example of Convolution Operation with Zero Padding

To suit the kernel size, it adds zeros to the rows and columns on either side of the tensor matrix. In Figure 2.6, for example, the input tensor 5x5x1 and kernel 3x3x1 provide an output of 3x3x1, reducing the spatial dimension. However, using the zero padding strategy of inserting zeros, we get a 5x5x1 output feature value, preserving the dimensionality.

The following formula is used to approximate the output dimension value to understand the relationship between the input and output image dimensions.

$$O = \frac{(i - F + 2P)}{s} + 1 \quad (2.1)$$

Here, i stands for input size, F is the filter size, P is the padding, and S is the stride value used. So, if the input image dimension is 32x32x3, the filter size is 3x3x3, and the stride is 1, we add 10 of these filters, and the final output dimension is $((32-3+0)/1) + 1 = 30$. As a result, the output image size is 30x30x10. One of the biggest advantages of CNNs

vs NNs is that you would not have to flatten the input images to 1D as CNNs are capable of handling 2D image data. The next section discusses different elements of Convolutional Neural Network.

2.3.1 Input Image

An input image is a representation of pixels with differing pixel strengths, which are calculated in pixel intensity. Any dimension values given as Height (H) \times Width (W) \times Channel (C) can be used as the input image size. Since the CIFAR-10 dataset will be used for training and validation, the input image will be $32 \times 32 \times 3$. Working with images has a range of disadvantages, including the inaccuracy of features that a human eye may detect and the necessity for thorough research to extract patterns through these images as converted to data for machines. Greyscale, RGB, CMYK, and other input channel matrices may be used in some situations.

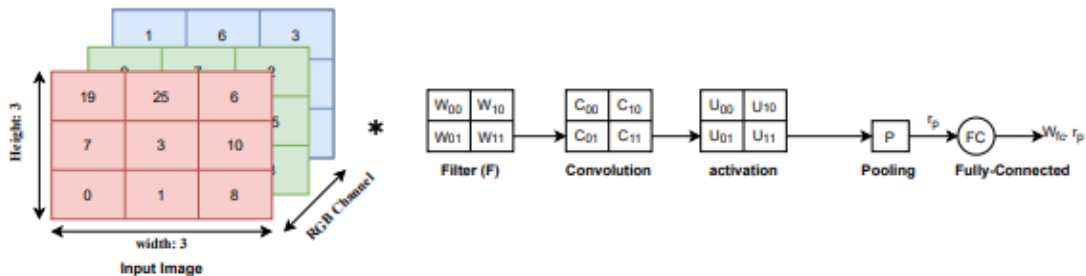


Figure 2.7. Network Input: RGB Image

However, there are three input channels in this case: red, green, and blue (RGB). The reason we choose RGB is that it has been shown that combining these three colors will yield any color palette. And, in most cases, we are concerned with colored images made up of several pixels, each of which is made up of different RGB channel values. Three RGB channels with height units of 3 and width units of 3 are shown in Figure 2.7.

2.3.2 Convolutional Layers

The key block of every CNN model is the convolution layer, which consists of various combinations of linear and non-linear operations and performs the task of feature extraction. Any neuron in this convolution layer has a receptive field that communicates with adjacent neurons in the previous layer through a series of trainable weight values. Every time a new feature map is computed, the input is convolved with the learned weight value and sent by non-linear activation.

2.3.3 Pooling Layers

The pooling layer sits between various convolution layers, and its main objective is to minimize the input's spatial dimensions while maintaining the channel/depth dimension unchanged. It is also known as down-sampling because it decreases the image size while preserving the most relevant details. Max pooling, average pooling, and adaptive pooling are the most common techniques, which slide a fixed window over the layer and select one raw value depending on the type of pooling used. To reduce computations and overfitting risks, max pooling selects the largest of all units and down samples weights, while average pooling selects the average of all units.

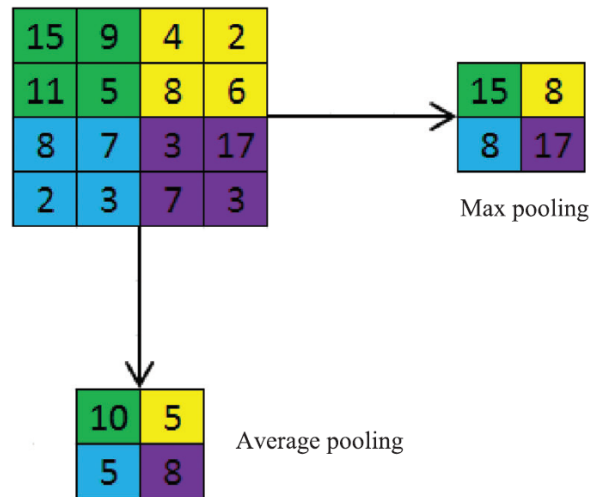


Figure 2.8. Pooling Methods

2.3.4 Activation Layers

Activation functions add a non-linear transformation to the given input, allowing it to learn and do better on more complex tasks, and thereby aiding NN in learning more complex non-linear datasets. If these functions are not used, the machine produces linear functions of degree one, which are simple to solve but have a lower non-linearity for learning complex objects like speech, video, or an image. The final layer activation function differs from the others in that it assists in the normalization of the output true values to the intended class probabilities.

2.3.5 Fully Connected Layers

The fully connected layer, which executes sorting, is the final layer of CNN architecture. All the features from the previous convolution and pooling layers are converted into a 1D list of integers and transferred into these fully connected layers with an activation mechanism. They are also known as dense layers because they use learnable weights to map any input to an output. Many comparisons have been made using various classifiers, and this field also needs further study, and it remains a research focus.

2.4 Gradient Descent

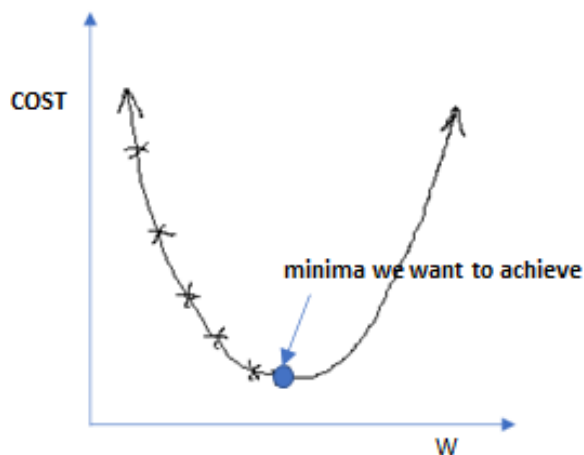


Figure 2.9. Theory Behind a Local Minimum

The task of minimizing the cost of the object function $J(\theta)$ by the model parameters, where θ belongs to \mathbb{R}^d is known as the method of optimization, or ‘Gradient Descent’ in the field of ML/DL. It basically determines the change in weight value because of the change in error. It can be visualized graphically as seen below, with the aim of achieving a steeper slope such that our model can learn faster and reach its global minima value [23].

With respect to its input, it is nothing but the partial derivative of that input. Consider a basic linear model in which the error (E) is given as:

$$E = Y_p - Y_a \tag{2.2}$$

Here Y_p is the predicted value and Y_a is the actual value. The key goal of these machine learning models is to help minimize errors and provide optimum accuracy. The model’s efficiency is also influenced by the learning rate value (Lr); if the Lr value is low, the model takes longer to learn; if the Lr value is high, the model takes less time to learn.

2.5 Parameter Calculations

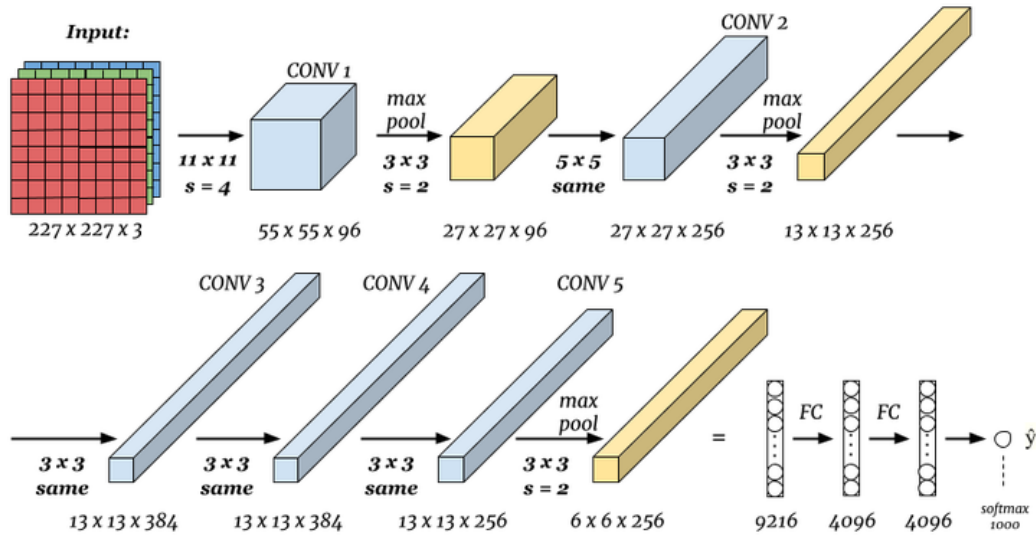


Figure 2.10. Illustration: Calculations of the Number of Parameters

Table 2.1. CNN Parameter Calculations

Operator Layer	Tensor Size	No. of learnable parameters
Input	$227^2 \times 3$	0
Conv1	$55^2 \times 96$	34944
MaxPool1	$27^2 \times 96$	0
Conv2	$27^2 \times 256$	614656
MaxPool2	$13^2 \times 256$	0
Conv3	$13^2 \times 84$	193620
Conv4	$13^2 \times 384$	1327488
Conv5	$13^2 \times 256$	884992
MaxPool3	$6^2 \times 256$	0
FC1	$1^2 \times 4096$	37752832
FC2	$1^2 \times 4096$	16781312
FC3	$1^2 \times 1000$	4097000
Output	$1^2 \times 1000$	0
Total	-	61, 68, 6844

Our optimal target with CNN is to learn the values of the filter using the back propagation mechanism. The count of learnable elements within a layer is represented by the number of parameters. There are a few main mathematical expressions that are needed to comprehend the final parameter calculations. The output shape is given as listed by the equation 2.1. For example, as in Figure 2.10, we have input as $227 \times 227 \times 3$, where $i=227$, $k=11$, $s=4$, $p=0$, therefore the output shape will be $55 \times 55 \times 96$, where 96 is the number of channels.

$$O = \frac{(227 - 11 + 2(0))}{4} + 1 = 55$$

In CNN, there is little to learn in the input and pooling layers, so there are no learnable parameters after these layers. The parameters learned after each convolution layer are $[(m*n*d) + 1] * k$ for width m , height n , d for the number of filters in the previous layer, and the number of filters k . In addition to that the parameters after each FC Layer are calculated as $[(n_c * n_p) + n_c]$, where n_c stands for current layer and n_p stands for previous layer and c is the current layer channels. A sample CNN architecture for the calculation of the number of parameters is covered as in Figure 2.10, Here this consideration was done for ImageNet dataset, so the output layer gives probability for 1000 classes.

2.6 Training Process of CNNs

The training was done on the CIFAR-10 labelled dataset, which was partitioned for training and testing. This method moves forward or backward to find the right values for weights and kernels at each point, assisting in the cost function minimization. The model detects the error in forward propagation and uses gradient descent to give modified weights back to the user. The performance layer error is estimated to help CNN learn more effectively and produce correct class probabilities.

$$error = \sum \frac{(P_{targetclass} - P_{output})^2}{2} \quad (2.3)$$

Larger data is supposed to be fed to these CNN models for improved performance, but application limits and size management for edge embedded applications must be considered. Backpropagation is the most common algorithm used in the training phase of a CNN model, and it proceeds until a global minima value is reached.

2.7 Visualization

The key concept behind visualization is to find out which characteristics of the input are retained in the feature maps. A very detailed or Fine-grained information should be obtained from the data, while feature maps should capture more generalized features. The following Figure 2.11 gives a general illustration of how the input is preserved after every layer in a CNN model and how it generalizes features to fit in any of the output classes [24].

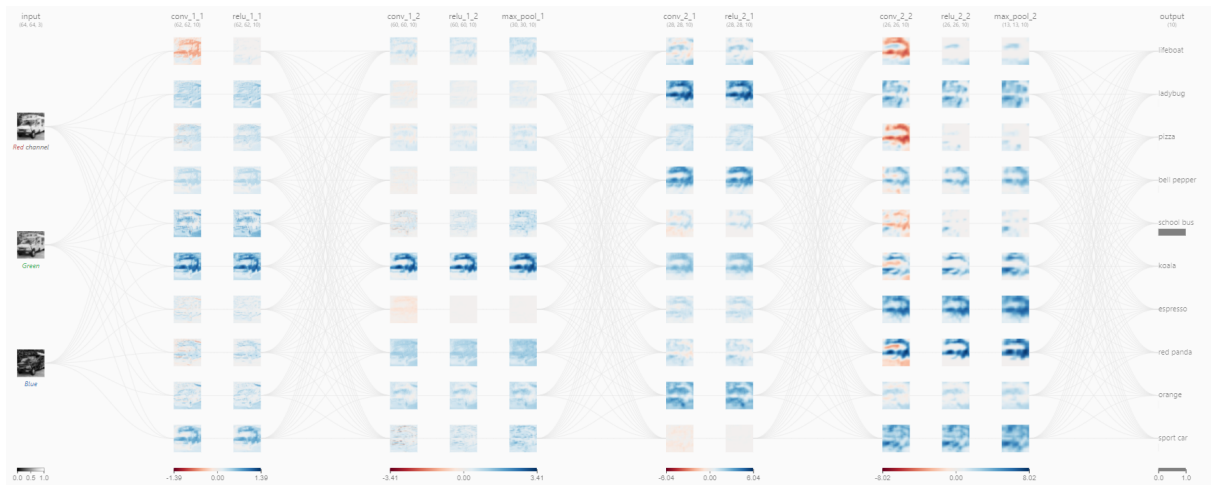


Figure 2.11. CNN Visualization (Input: School bus)

For each layer, new features are learned; for example, some layers will focus on pixel borders, while others will focus on intensity, while others will focus on various shapes, and so on. The final layer is a completely linked layer that contains a category of various groups that are used in this model for classification purpose. In this case, we have fed a school bus as an input, which passes through all these layers and yields a probability of 0.9947 for the school bus class.

2.8 Measure of Performance

The measure of performance of a CNN model is its ability to give close to better predictions, referred to as model accuracy, its success using transfer learning techniques, the time it takes to train on various machines such as TPUs, GPUs, or CPUs, and how fast the training process is in general (model speed).

2.9 Related Architectures

2.9.1 HBONet: Harmonious Bottleneck Network (Baseline)

Several neural network architecture variants have appeared in recent years, with an emphasis on mobile edge applications. This section summarizes the previously discovered light models and focuses solely on transformational approaches of spatial measurements, as well as including a description of the inverted residual technique. Depth-wise separable convolutions are used in the Harmonious Bottleneck approach to concentrate on both the spatial and channel measurements. This method is divided into two sections: first, down-sampling the spatial dimension while maintaining the channels steady ($H/s \times W/s \times C_1$), and then extending the channels ($H/s \times W/s \times t \times C_1$), and second, up-sampling the spatial dimensions while halving the channel reduction ($H \times W \times C_2/2$), and finally, concatenating with the partial channels of the input tensor ($H \times W \times C_2/2$) or its pooled version. The final calculated value of this module is:

$$Cost = \frac{B}{s^2} + \left[\left(\frac{H}{s} \times \frac{W}{s} \times C_1 \right) + (H \times W \times C_2) \right] \times K \times K \quad (2.4)$$

Here the kernel size is denoted by K , and the measured value of the embedded blocks between the two operating parts is denoted by B .

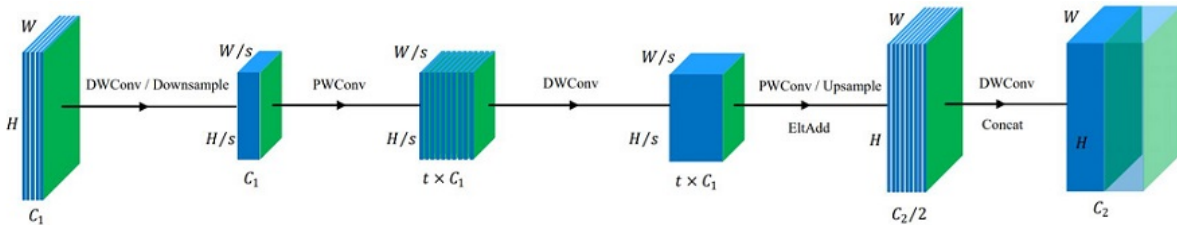


Figure 2.12. Operational Bottleneck Block of HBONet

If used in some CNN architecture, this implementation produces lighter models with impressive precision [25]. The Harmonious Bottleneck extended to two Orthogonal Dimensions (HBO) is made up of two parts: spatial (H/W) contraction-expansion and channel (C) expansion-contraction, all of which are arranged in a bilaterally symmetric form as seen in

Figure 2.12, and work in harmony. The following are the strategic measures that were taken to form the HBONet bottleneck module:

- The depth-wise convolution (DWConv) approach is used to downsample the input.
- Using the point-wise convolution (PWConv) operation, the number of channels is compounded by an expansion factor of ‘t’ while the spatial components are preserved.
- A later stage uses DWConv and PWConv to upsample and execute element-by-element addition (EltAdd).
- Finally, the partial channels of the input are concatenated.

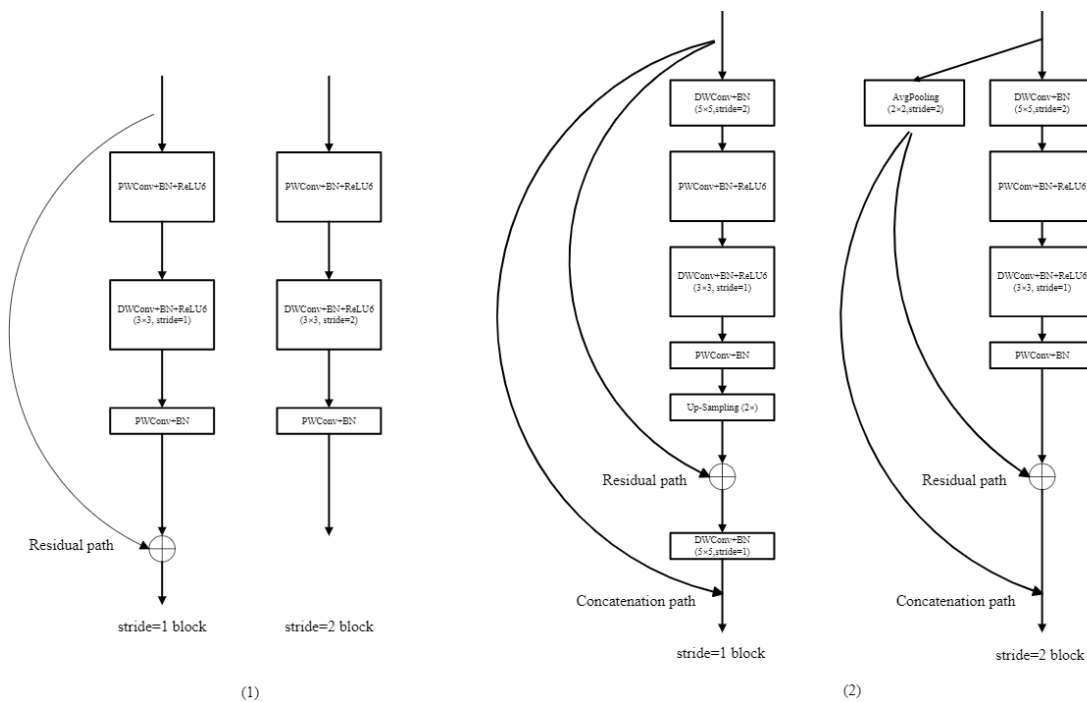


Figure 2.13. Inverted Residual with Linear Bottleneck and Harmonious Bottleneck, with strides

This work is heavily inspired by the bottleneck module of MobileNetV2, where orthogonal space appears to be unexplored. A residual path is used in addition to the techniques described above to help with gradient propagation and to implement this modern age

Table 2.2. HBONet Architecture

Operator Layer	Input Size	t	c	n	s
conv2d 3×3	$224^2 \times 3$	-	32	1	2
Harmonious Bottleneck	$112^2 \times 32$	1	20	1	1
Harmonious Bottleneck	$112^2 \times 20$	2	36	1	1
Harmonious Bottleneck	$112^2 \times 36$	2	72	3	2
Harmonious Bottleneck	$56^2 \times 72$	2	96	4	2
Harmonious Bottleneck	$28^2 \times 96$	2	192	4	2
Harmonious Bottleneck	$14^2 \times 192$	2	288	1	1
conv2d 1×1	$14^2 \times 288$	-	144	1	1
Inverted Residual	$14^2 \times 144$	6	200	2	2
Inverted Residual	$7^2 \times 200$	6	400	1	1
conv2d 1×1	$7^2 \times 400$	-	1600	1	1
avgpool 7×7	$7^2 \times 1600$	-	-	1	-
conv2d 1×1	$1^2 \times 1600$	-	k	-	-

lightweight model. The concatenation operation is crucial because it decreases the number of output channels that must be computed and encourages the reuse of feature information. The original baseline architecture of Harmonious Bottleneck (HBONet) can be seen in Table 2.2.

2.9.2 Flipped Inverted Residual (FIR): Sandglass Structure

As part of this analysis, the principles of residual skip connection are clarified. There have been a few basic questions asked about the placement of this structure in a network, such as, (i) the impact of placing them at higher dimensions, (ii) could knowledge be lost if linear activations are added to bottlenecks, and (iii) cost savings by replacing dense spatial convolutions with depth-wise ones to further minimize computational complexity; however, the issue of whether this depth-wise convolution can be applied to the lower dimension remains unanswered. The study discussed in [26], was so inspired by these questions that it invented a modern bottleneck design called the sandglass block. Due to the constraints of lower dimensionality at the start of inverted residual blocks, which is expected to impede the efficient capturing of useful information due to channel compression, we use the sandglass block technique, which has a wider architecture and is expected to minimize gradient uncer-

tainty, as stated in a recent study. We effectively combine these two concepts and discuss user space modifications in this work.

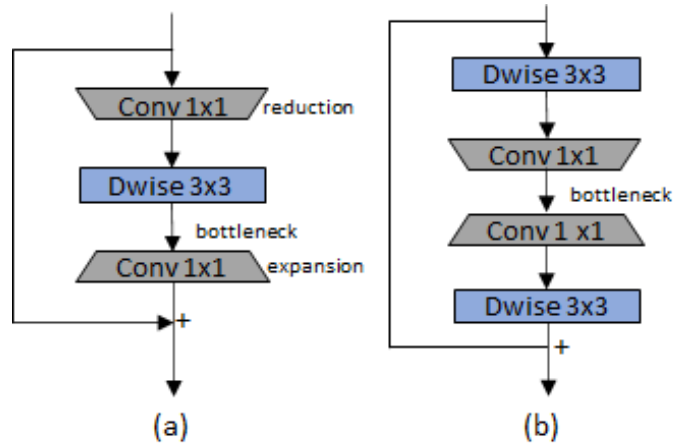


Figure 2.14. Residual blocks (a) Traditional Bottleneck Arrangement, (b) Block Sandglass Bottleneck Arrangement

The sandglass block was designed to shield more component details when it transitions from lower to upper layers, allowing residual connections to bind higher dimension features. To extract rich spatial information, it uses lightweight 3 x 3 depth-wise convolutions applied to the higher dimensions. In conclusion, this article made the following contributions:

- For mobile edge development, rethinking the bottleneck structure modifications.
- According to the report, if this technique is used and they are put in high-dimensional fields using depth-wise convolutions, it encourages improved learning and model efficiency.
- A new research is proposed that significantly expands the traditional bottleneck structure and is better suited for mobile launch.

3. HARDWARE AND SOFTWARE

The key concept is that the use of CIFAR-10 dataset to train our HBONext model for image classification. CIFAR-10 is a collection of 60,000 images (32 x 32) separated into ten groups that is widely used in deep learning and computer vision applications. This dataset is further divided into two sections for training and validation to better understand the model's results.

- Nvidia GTX 1080Ti GPU
- Google Colab environment
- BlueBox 2.0 hardware by NXP
- PyTorch 1.0+ Framework
- Python 3.6.7 version
- Spyder 3.6 version
- RTMaps Intempora
- RTMaps 4.0.
- RTMaps Intempora
- Livelossplot (Loss and accuracy visualization)
- Teraterm/putty to visualizing the output

3.1 Hardware Used

The whole model training was completed in the Google Colab environment, an easy-to-use platform that allows Google servers free access to any available GPUs. The NVIDIA GeForce GTX 1080Ti GPU is also used to produce the initial results. For graphical representation and calculations of the number of parameters, PyTorch-based packages such as Livelossplot and torchsummaryX were used.

3.1.1 NXP Bluebox 2.0 (BLBX2)

NXP's BlueBox2.0 [27], [28] is a real-time development platform that allows self-driving cars to follow accuracy, technical safety, and automotive durability standards. It is a one-stop shop for creating self-driving apps, such as ADAS and driver assistance systems. The perception S32V234, sensor microchip, the LS2084A added to the PC CPU, and a radar microcontroller S32R27 are the three devices on chip. It runs on a different Linux OS embedded on both the LS2 and S32V processors using RTMaps. That is the system's central processing unit. Power, S32V RST, S32R RST, and LS2 RST buttons are located on the front side, along with 10GEP1 Ethernet, 10GEP3 Ethernet, USB, SD card slots for LS2 and S32V, UART, and HDMI link slots.



Figure 3.1. BlueBox 2.0 by NXP

As a result, CNN/DNN models would be more robust and efficient in the ADAS context. This BlueBox 2.0 by NXP is a software framework with practical protection, vision acceleration, and vehicle interfaces for Autonomous Drive and Sensor Fusion applications. The Society of Automotive Engineers (SAE) has suggested criteria for the degree of autonomy, and while there are many determining factors, here are a few that will drive the future of autonomy.

0th Level: The driver must keep a close eye on the car and perform all the required duties such as accelerating, steering, and braking. With today's speed, these cars are expected to have blind spot detection, crash warning signs, and an auto-emergency braking system.

1st Level: At this level, the automation system progressively takes over control of the car, incorporating functions such as adaptive cruise control, which manages braking and acceleration.

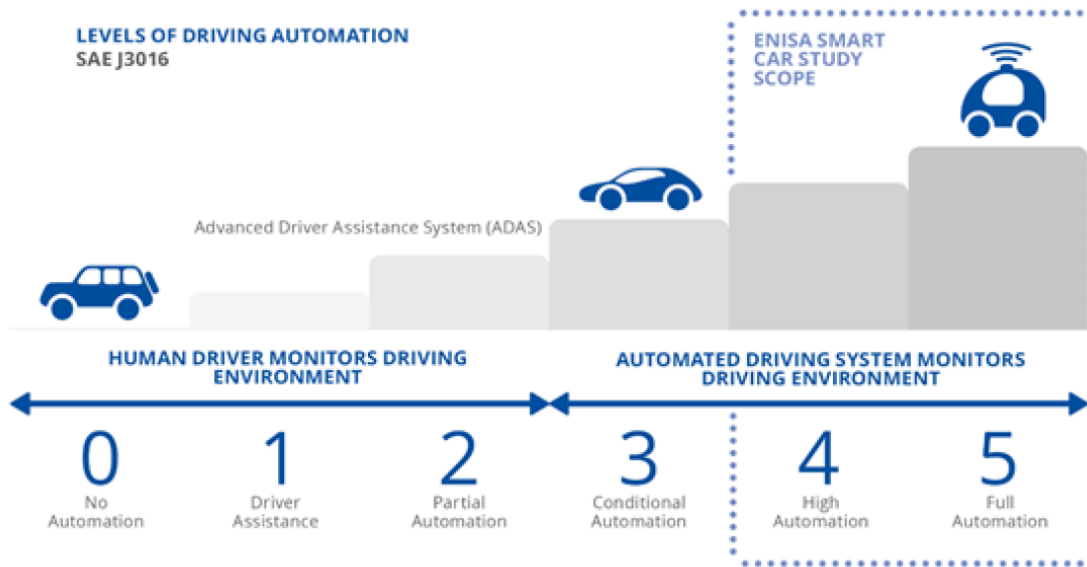


Figure 3.2. Levels of Autonomy

2nd Level: At this level, the autonomous system can perform more complex tasks such as lateral and longitudinal controls while retaining a high level of perception of its surroundings. This level includes features such as traffic assistance and the option to take your hands off the wheel and breathe for a moment.

3rd Level: For specific vehicle speeds, routes, and conditions, drivers can fully disconnect from driving. In a traffic-like scenario, the user can do other things as the system warns them to retake control. It also means that if the driver is not paying attention, the vehicle can continue to a safer location.

4th Level: The system is entirely capable of tracking the ambient atmosphere and regulating all driving functions under various situations. If those requirements are not met, the machine will notify the customer.

5th Level: A car at this level is capable of maximum autonomy. There is no need for a driver behind the wheel, and often there is no need for a steering wheel, brakes, or pedals. It is expected that a cabin would be available for travelers who want to fly from one place to another. It is supposed to understand voice commands and monitor various features by voice, like today's Siri, Alexa, and Google Assistant.

3.1.2 Features and Specifications of BLBX2

- It has various interfaces for vehicle I/O
- Automotive interfaces with perception acceleration using ASIL-B compute
- Dedicated interfaces for the ASIL-D subsystem.
- 12 V/24 V power units
- 8x cameras, Ethernet 100M/ 1G/ 10Gbps, SFP+, 8x 100BASE-T1, CAN-FD
- 16 GB DDR4 and 256 GB SSD for high performance computing
- Vehicle vision and sensor fusion processor, S32V234
- Integrated compute machine LS2084A
- Radar microcontroller S32R27
- Technologies like CSE and ARM by TrustZone
- ROS Space, a Linux-based architecture based on ROS
- Programmable in linear C and conveniently customizable
- Radar, perception, LiDAR, and V2X data streams

Figure 3.3 displays the architectural representation of BlueBox 2.0, which consists of LS2 and S32V independent processors. BlueBox serves as the system’s central computing unit, allowing for quick DNN model deployment and support for a variety of ADAS applications. The architecture of this combined board shows that it supports various communication protocols such as UART, CANFD, FlexRay, JTAG for interfacing, and on-board memory interface.

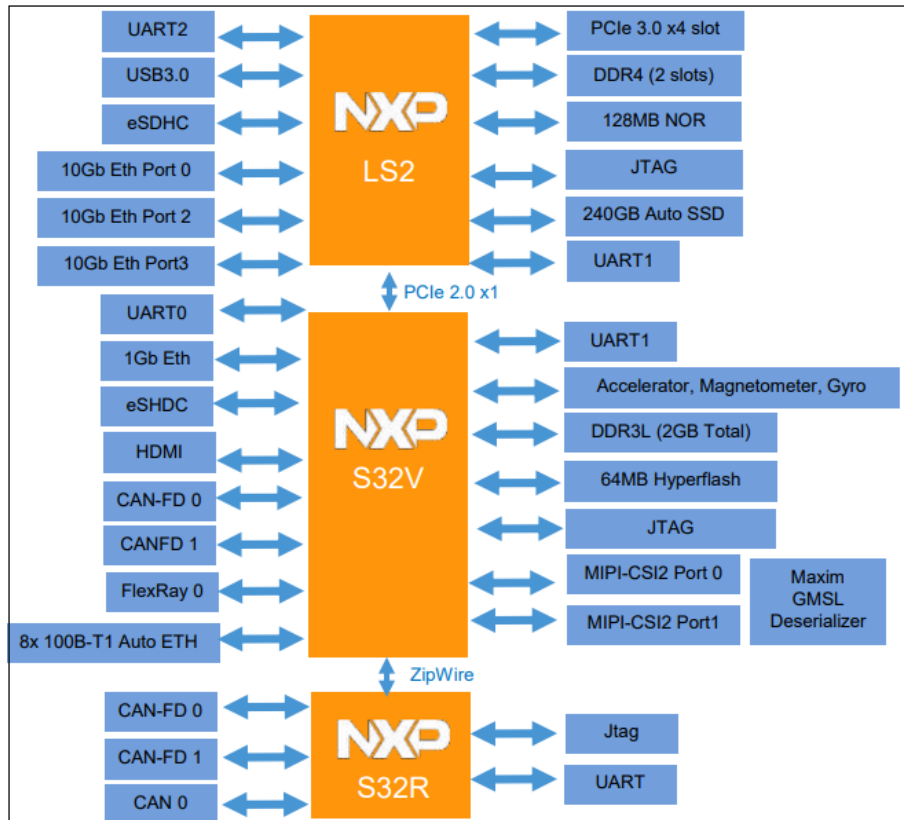


Figure 3.3. General Architecture of BLBX2

3.1.3 Vision Processor – S32V234

An efficient 3D GPU, ISP, vision accelerators, safety modules, dual APEX2, camera, highly capable ADAS, object detection and classification competency, image processing platform, Machine Learning, and Sensor Fusion implementations are all part of the S32V234 unit’s device architecture. This system processor has MIPI-CSI camera inputs, which aids in the integration of several cameras for image conditioning. It is a second-generation vision processor with 32-bit ARM Cortex A56 S32V, Cortex A53 cores, and an ARM M4 core on chip, and has development studio for user-modified applications [29]. It runs on the Ubuntu 16.04 operating system and is driven by an SD card with interfacing for users in front of the BLBX2.

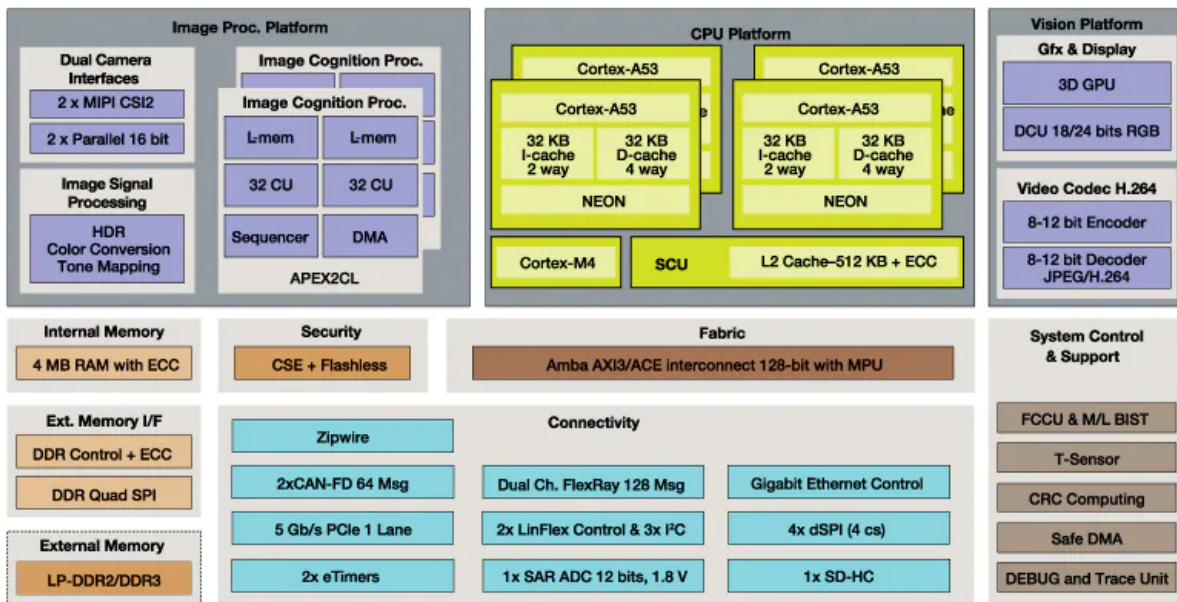


Figure 3.4. S32V234 Block Diagram

3.1.4 Processor - LS2084A

The LS2084A contains two 64-bit ARM Cortex A72 CPUs, two 64-bit DDR4 SDRAM memories, ethernet interfaces that support up to eight 10 Gbps, or sixteen 1/2.5 Gbps MACs, hardware virtualization, and other connectivity peripherals for possible expansion. As a result, it is well-suited to high-performance applications. It also includes an SD card that enables this processor to run Ubuntu 16.04 on BlueBox as an OS4.3 platform [30]. This processor chip can be used for a variety of tasks, including integrated control, router application layer processing, switches, gateways, general purpose embedded computing systems, and ADAS.

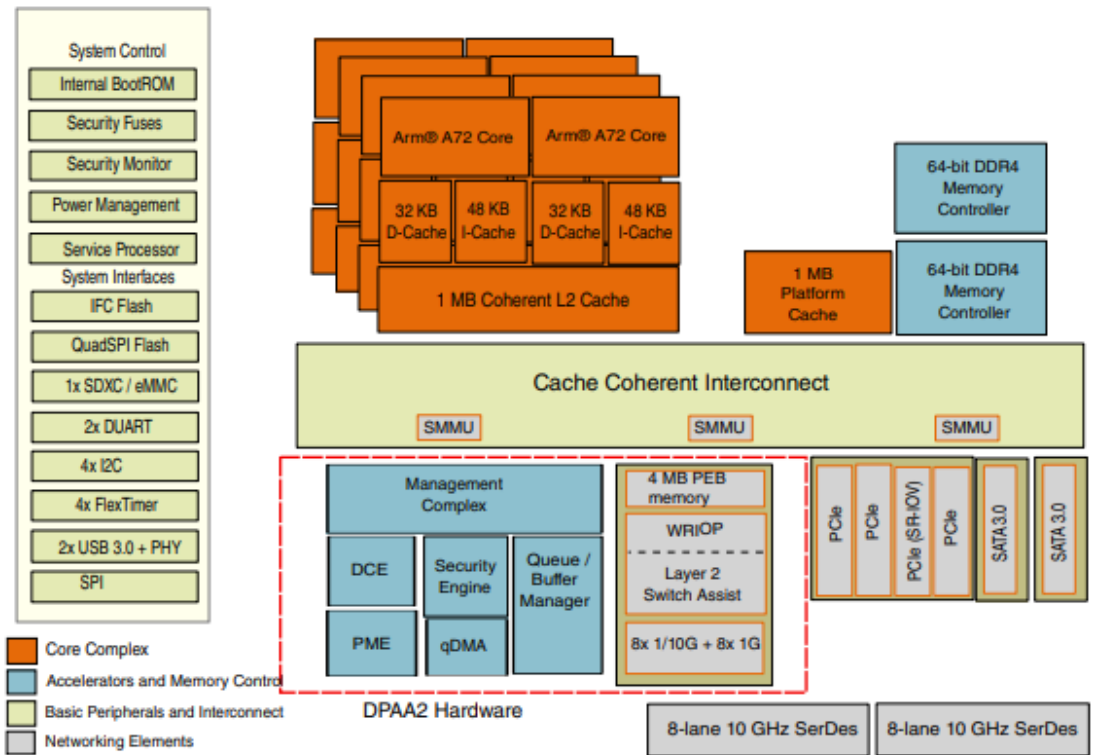


Figure 3.5. LS-2084A Block Diagram

3.2 Software Used

3.2.1 Real-Time Multi Sensor Application (RTMaps)

RTMaps is an asynchronously more efficient tool that comes with a powerful and simple-to-use framework for fast and stable development. It enables the creation, assessment, validation, benchmarking, and execution of multimodal dependent applications, as well as the integration of multiple sensors such as camera, lidar, and radar. The RTMaps Embedded version contains a ‘Remote Studio’ feature that runs on the targeted edge hardware and manages execution from a desktop machine running Linux or Windows connected over the network. The RTMaps embedded platform contains the runtime engine and component libraries needed to run any ARM or x86 platform, such as the Raspberry Pi, BLBX2, MicroAutobox by DSpace, and others.



Figure 3.6. RTMaps Remote Studio connected to BLBX2

TCP/IP networking is used to create a connection between RTMaps studio on a desktop computer and BLBX2. Its component library includes software modules that support packages like C++, Python, Simulink, and 3-D vision, among others, making deployment and interfacing easier.

3.2.2 Various Deep Learning Frameworks

This area of AI is encouraging businesses to create sophisticated and devoted software that is customized according to the user/customer to provide smart solutions. This is one of the reasons for the development of various deep learning frameworks, which are open-source libraries or tools.

TensorFlow [31] is a popular and commonly used deep learning platform developed by the Google Brain team, initially released in 2015, and it is available as a DLL for mobile and desktop applications. For the implementation of deep neural network models, it has a support group designed for C++, Python, and R.

Tensorboard is a visualization toolkit of TensorFlow that converts data for visualization purposes to see model performance or perform network modeling. TensorFlow Serving is a service that helps you to easily install architectures and connect with other models. There are several advantages of using Tensorflow, including outstanding documentation and community support since it is Python-based, solid GPU support, a stronger visualization toolkit, and so on.



Figure 3.7. DLL Frameworks

Keras [32] was created in 2015 by Francois Chollet of the Massachusetts Institute of Technology (MIT), and it has a long list of contributors. Keras libraries provide support for higher-level neural networks with a Python API, and it is worth noting that keras runs on top of TensorFlow, CNTK, and Theano as base frameworks. It is a lightweight, simple to use tool that is widely used for classification, speech recognition and translation, text generation, and other applications. It also has built-in support for multi-GPU parallelism and training.

PyTorch [33], [34], [35], was created by Facebook's AI Research Lab's Adam Paszke, Soumitha Chintala, Sam, and Gregory. Python, C/C++ libraries for some processing, and CUDA were all included. It is also known as 'Torch', and it is widely used in the deep learning community. It strongly encourages fast prototyping and parallelism using multiple GPUs. Google, Twitter, and Facebook are among the organizations that have made heavy use of PyTorch. This thesis study was entirely implemented using PyTorch as a basis for implementing our updated deep CNN models (DCNNs).

4. DESIGN EXPLORATION AND MODIFICATIONS

This chapter discusses numerous design space adjustment strategies that have assisted in achieving the goal of better model performance and its handling on the resource limited hardware. The proposed architecture HBONext: an HBONet variant with Flipped Inverted Residual block is introduced near the end of this segment.

4.1 Techniques to Improve Performance

To continue, a detailed review of the strategies that have benefited deep CNNs (DCNNs) in improving their performance has been conducted. Here are a few pointers that have assisted this thesis work in achieving the model accomplishments mentioned above in contrast to its baseline architecture:

- Data augmentation [35], a superior feature collection method for preprocessing data.
- Model hyperparameter tuning and tweaking, such as using different learning rate values, different activation functions, an optimizer, and scheduling strategies to improve model predictive performance.
- With the help of previous architectures and techniques, we were able to modify the current structure by including newer bottleneck designs and convolution layer with resampling and restacking that suited our needs.

4.2 Use of Optimizer

To minimize network errors and make as reliable predictions as possible, optimization algorithms are used to adjust simple attributes including learning rate and weight of our neural network models. It is governed by the following updating equation, which reduces the cost or losses:

$$W_{new} = W_{old} - L_r \times (\nabla wl) \times W_{old} \quad (4.1)$$

As seen in Figure 4.1, SGD, RMSPROP, ADAM, ADAGRAD, and ADADELTA are compared, with this graph illustrating how these various approaches converge. A graph of the number of iterations to the losses.

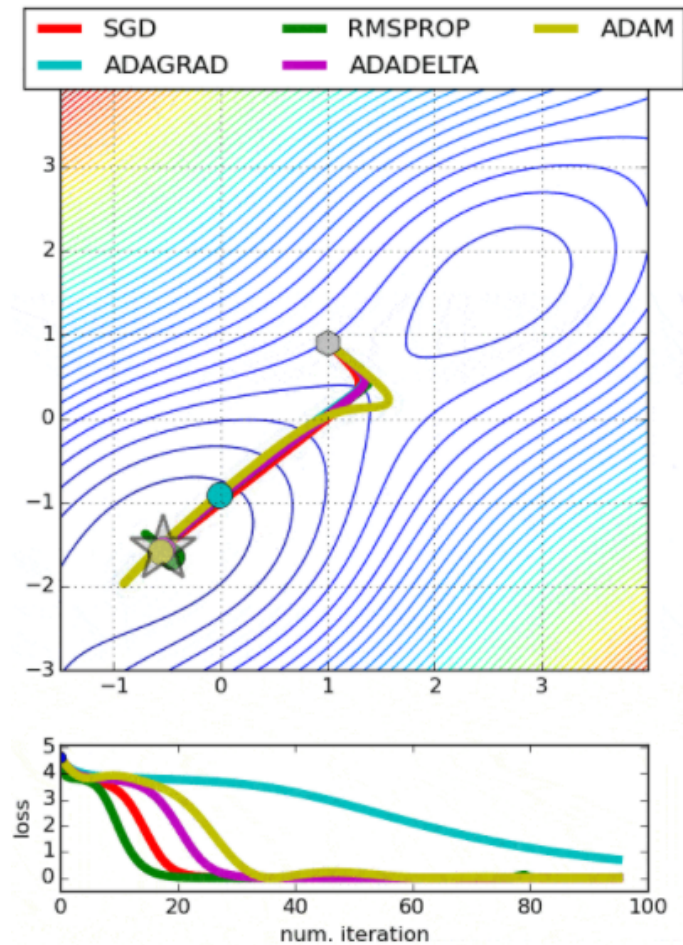


Figure 4.1. Comparison Between Different Optimizers

The PyTorch package `torch.optim` is used to name the optimizer in our training phase. With its periodic updating existence, the issue with SGD is that it generates high variance values, which has an indirect effect for convergence to a minima value. However, this “ravines problem” can be overcome with a benefit of momentum. We also comprehend the use of `nesterov` since it helps in the acceleration of SGD feature in the most effective way. We use stochastic gradient descent (SGD), which converges quicker on large datasets than gradient descent due to its frequent updates. Out of many optimizers, SGD is expected to use least

memory, so this work implements SGD with momentum value and nesterov in the proposed model.

Syntax example: `optimizer = torch.optim.SGD(model.parameters(), lr=0.1, momentum=0.9, nesterov = True)`

4.3 Learning Rate Scheduling Methods

The previous section of this chapter was focused on minimizing losses by optimizing algorithms based on the weight vector updating theorem but changing the learning rate is just as essential as the optimizing algorithm. There are several factors to consider, such as the degree of the learning rate, the rate of decay, proper initialization, and so on.



Figure 4.2. Performance of different L_r Schedules

With pre-defined schedules such as step decay, time dependent decay, cosine annealing, reduce plateau, and exponential decay, these learning schedules often change the value of Learning rate (L_r) while the training is in progress. So many options are proposed in `torch.optim.lr` scheduler to change the learning rate depending on the number of epochs. It is worth noting that Learning rate scheduling can only be used after the optimizer syntax has been declared in the train code. To compare, we use `CosineAnnealingLR` and `ReduceLROn-`

Plateau on the learning rate scheduling implementation. Out of which, ReduceLROnPlateau allows reduced L_r if no improvement is seen after a certain number of epochs, which eventually reduces the value of L_r . Our work mainly uses CosineAnnealingLR which has helped in the implementation of the proposed model architecture.

4.4 Use of Activation Function

Activation Functions are often used in models to add non-linearity and assess the relationship between the input and output signals. There are two kinds of activation mechanisms. Activation functions that are linear and non-linear. However, it can be mathematically shown that only a non-linear function can enable the network to learn in response to the random encountered errors. The use of ELU activation is considered for the HBONext v 1.1 since it can focus on the positive values only, to conserve the negative values with reduced computing expense. This is expressed as:

$$\begin{aligned}
 f(x) &= x, & \text{for } x > 0 \\
 &= \alpha(e^x - 1), & \text{for } x \leq 0
 \end{aligned}
 \tag{4.2}$$

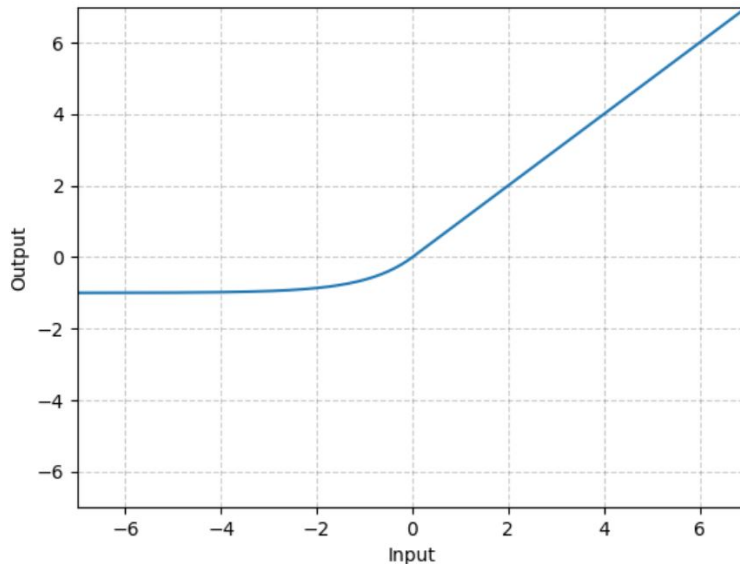


Figure 4.3. ELU non-linear Activation Function

It has alpha, which is a positive constant value typically used between 0.1 and 0.3. ELU helps generate more accurate results by converging to zero faster but cannot overcome exploding gradient problems [36]. This work also summarizes use of Mish activation, which is a new kind of activation function that is a gated softplus function. We also use Swish to later see the effect of this activation on the network, See the graphical representation of these activation functions in Figure 4.4. Besides, Mish function has helped avoid saturation because of near-zero gradient, the smallest negative gradients, good regularization, efficient optimizing, and generalization [37].

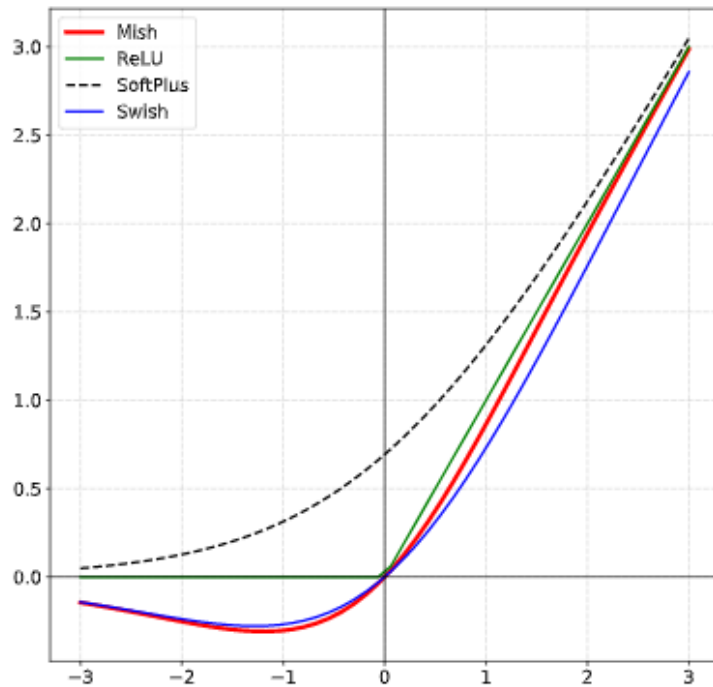


Figure 4.4. Activation Functions for Mish, ReLU, SoftPlus, and Swish

4.5 Checkpoint Save/Load Method

The aim of saving the checkpoint file is to test the proposed architecture’s later implementation capability on the BLBX2 embedded hardware platform. The `torch.save()` and `torch.load()` syntax are used in PyTorch to save and load checkpoint files which are mostly of the format `.ckpt`, `.pkl`, or `.pth`.

5. PROPOSED HBONEXT ARCHITECTURE

The derived harmonious bottleneck structure is discussed in this chapter (DHbneck). As seen in Table 2.2, it replaces the baseline’s original inverted residual block with a flipped inverted residual (FIR), resulting in a new light-weight architecture known as HBONext. In the Table 5.1 below for HBONext v1.1, the expansion factor is t , the channel output is c , the number of times the block repeats are n , and the stride value is s . Here are a few simple methods that were used in its implementation.

- With ELU in place, the non-Linear activation mechanism from Relu6 is carefully substituted.
- The addition of an element-by-element skip relation assists in the resolution of vanishing gradient issues. In addition, unlike the baseline architecture, the FIR block is modified as Dwise-Pwise-Pwise-Dwise to obtain a sandglass-like structure.
- Reconsidering the bottleneck module in terms of its spatial and channel measurements to help reduce the model size much further.

Table 5.1. HBONext v1.1 Architecture

Operator Layer	Input Size	t	c	n	s
conv2d 3×3	$32^2 \times 3$	-	64	1	1
FIR block	$16^2 \times 64$	2	32	1	1
DHbneck	$16^2 \times 32$	2	16	2	1
DHbneck	$16^2 \times 16$	2	32	4	2
DHbneck	$8^2 \times 32$	2	64	4	2
DHbneck	$4^2 \times 64$	2	96	4	2
DHbneck	$2^2 \times 96$	2	128	2	1
DHbneck	$2^2 \times 128$	1	256	2	1
conv2d 1×1	$2^2 \times 256$	1	512	1	2
FIR block	$2^2 \times 512$	2	256	1	2
FIR block	$1^2 \times 256$	2	128	1	1
FIR block	$1^2 \times 128$	1	10	1	1
conv2d 1×1	$1^2 \times 10$	-	1024	1	1
avgpool 7×7	$1^2 \times 1024$	-	-	1	-
FC Layer	$1^2 \times 1024$	-	k	-	-

5.1 Design Considerations

Design consideration 1: Derived Harmonious Bottleneck (DHBneck)

- By selecting the 3 x 3 kernel and the ELU activation function, a simple adjustment can be made. The block in Figure 5.1, demonstrates the HBONext bottleneck structure in depth, as well as the FIR skip relation with the required stride value range.

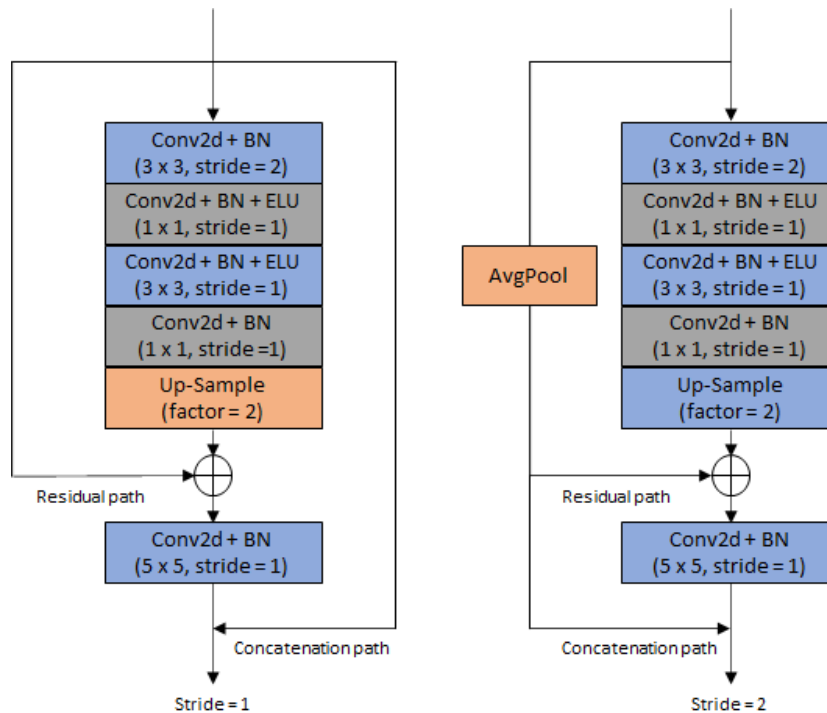


Figure 5.1. Harmonious Bottleneck Design with Different Strides

Design consideration 2: Flipped Inverted Residual Block (FIR)

- The sandglass block in Figure 2.14 (b), was designed to cover more component details when it transitions from the lower layer to the top layer, allowing residual connections to link higher dimension features.
- To extract rich spatial details, it uses lightweight 3 x 3 depth-wise convolutions applied to the higher dimensions.

Table 5.2. HBONext v1.2 Architecture

Operator Layer	Input Size	t	c	n	s
conv2d 3×3	$32^2 \times 3$	-	32	1	2
SepConv 3×3	$16^2 \times 32$	1	16	1	2
FIR block	$16^2 \times 16$	2	16	1	1
DHbneck	$16^2 \times 16$	2	24	1	1
DHbneck	$16^2 \times 24$	2	32	1	2
FIR block	$8^2 \times 32$	6	64	1	2
DHbneck	$4^2 \times 64$	2	64	1	1
DHbneck	$4^2 \times 64$	1	96	1	1
FIR block	$4^2 \times 96$	6	128	1	2
DHbneck	$2^2 \times 128$	2	192	1	1
FIR block	$2^2 \times 192$	6	288	1	1
conv2d 1×1	$2^2 \times 288$	-	1024	1	1
avgpool 7×7	$2^2 \times 1024$	-	-	1	-
FC Layer $\times 1$	$1^2 \times 1024$	-	k	-	-

This work also highlights the HBONext v1.2 that has helped greatly for our hardware deployment purpose with a slight change to the way layers are arranged and has helped in reducing the number of parameters, which in turn has helped for lower model size. Here we also use a technique of cosine annealing to achieve the model performance results and thus, the HBONext v1.2 can be seen in Table 5.2. The following modifications contributed to a further decrease in model size and improved performance with HBONext v1.2:

- Using extra Separable convolution at the input layer
- Identifying and resolving organizational bottlenecks
- Cutting down on the number of times, they repeat themselves
- Data augmentation for preparation using the mish activation mechanism, with Learning rate scheduling method Cosine Annealing

5.2 Width Multiplier Consideration

The goal of using different widths is to consistently thin the network at each layer. It is used to produce models that are smaller and less computationally costly. We can reduce the number of parameters and layer operations by using lower width values and propose reasonable solutions. In this work standard width multiplier values of 1, 0.75, 0.5, and 0.25 were used to implement the proposed model variants. Lower width values allowed us to reduce the number of parameters and layer operations, resulting in a reasonable smaller model with nearly identical output performance metrics and a smaller model size [38].

6. RESULTS AND DISCUSSIONS

The key concept was to use the CIFAR-10 dataset to train our HBONext model for image classification. CIFAR-10 is a collection of 60,000 images (32×32) separated into ten groups that is widely used in deep learning and computer vision applications. To understand the model success based on its accuracy parameter, this dataset is further divided into two sections for training and validation.

6.1 Model Accuracy

The whole model training was conducted in the Google Colab environment, an easy-to-use platform that allows Google servers free access to any available GPUs. The NVIDIA GeForce GTX 1080Ti GPU is also used to produce the later performance as in HBONext v1.2 (a), and HBONext v1.2 (b). For graphical representation and calculations of the number to parameters, PyTorch-based packages such as Livelossplot and torchsummaryX were used.

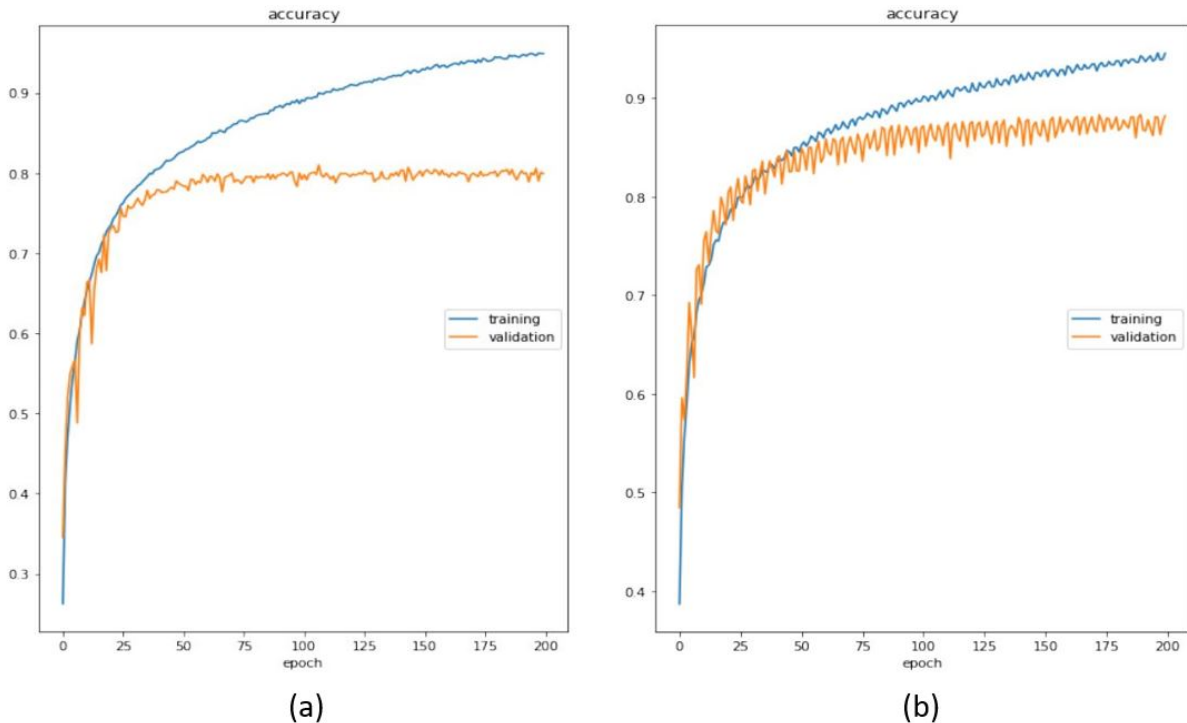


Figure 6.1. Accuracy vs the Number of Epochs: (a) HBONet (Baseline), (b) HBONext v1.1 (Proposed Architecture)

Table 6.1. Comparison: Total Number of Parameters

Model Variants	Number of parameters
HBONet (Baseline)	2.9586 M
Proposed HBONext v1.1	0.9458 M
Proposed HBONext v1.2 (a)	0.5255 M
Proposed HBONext v1.2 (b)	0.4733 M

We used standard width multiplier values of 1, 0.75, 0.5, and 0.25 to enforce our proposed model. The aim of using different widths is to uniformly thin the network at each layer. The entire model is trained using the Stochastic Gradient Descent (SGD) optimizer, with the momentum set to 0.9, the weight decay set to $4e-5$, and nesterov included. A batch size of 128 is used in the model, with a learning rate of 0.01. We were also able to obtain competitive outcomes by using a cosine annealing scheduler to set the learning rate. When compared to the baseline model, HBONet, the proposed model, HBONext v1.1, trained from scratch using CIFAR-10, improved precision by 12.13% percent while reducing model size by 65.18%. The use of harmonious bottlenecks in conjunction with the FIR technique has aided in the achievement of the following results.

6.2 Reducing Overfitting

There are 525.514k (HBONext v1.2) parameters in our theoretical neural network architecture version 2. The dataset classifies our model to predict 10 classes, and the mapping from the input image to the output class label shows that learning these many parameters may invite overfitting problems. Therefore, the following section addresses our approach to these issues [4].

6.2.1 Data Augmentation

This technique is commonly used to reduce the issue of image data overfitting. This data augmentation technique uses a proven label-preserving transformation to artificially expand the dataset [35]. In this implementation, we first generate these transformed images locally

on a CPU using a simple python script, then train our model on the GPU, making these approaches computationally free.



(a)



(b)

Figure 6.2. CIFAR-10 Dataset (a) without Augmentation, (b) with Augmentation

The generation of image input translations and horizontal reflections is the first approach to data augmentation, while altering the intensities of the RGB training images is the second method. In Figure 6.2, augmentation results on CIFAR-10 dataset can be seen.

6.2.2 Dropout

The output of each neuron with probability $p=0.5$ is set to zero in a dropout technique. To avoid overfitting, it is a regularization mechanism that deactivates a few neurons in the neural network forward pass at random and does not backpropagate it. Dropout has been suggested for use at the Fully connected layer in some research, and few studies have addressed its use on pooling layers. When it comes to our network, however, we use it at the end, at the FC Layer.

Table 6.2. Performance Comparison for HBONext for w=1.0

Model Variants	Accuracy	Model Size
HBONet (Baseline)	80.97%	12.00 MB
Proposed HBONext v1.1	88.30%	7.66 MB
Proposed HBONext v1.2 (a)	89.70%	3.00 MB
Proposed HBONext v1.2 (b)	89.88%	1.90 MB

After applying both the above-mentioned techniques from the section 6.2.1 and 6.2.2, during the training process of our HBONext v1.2 (a), we get improved performance metric accuracy of 89.70% and best model size of 3.0 MB for deployment purpose with augmentation, as seen in Figure 6.3. However, we get HBONext v1.2 (b), applying Learning rate scheduling technique called ‘Cosine Annealing’ with augmentation we get profiles as seen in Figure 6.4. The performance comparison for the baseline model HBONet and all the variants of HBONext done using our GPU resource can be seen in the Table 6.2.

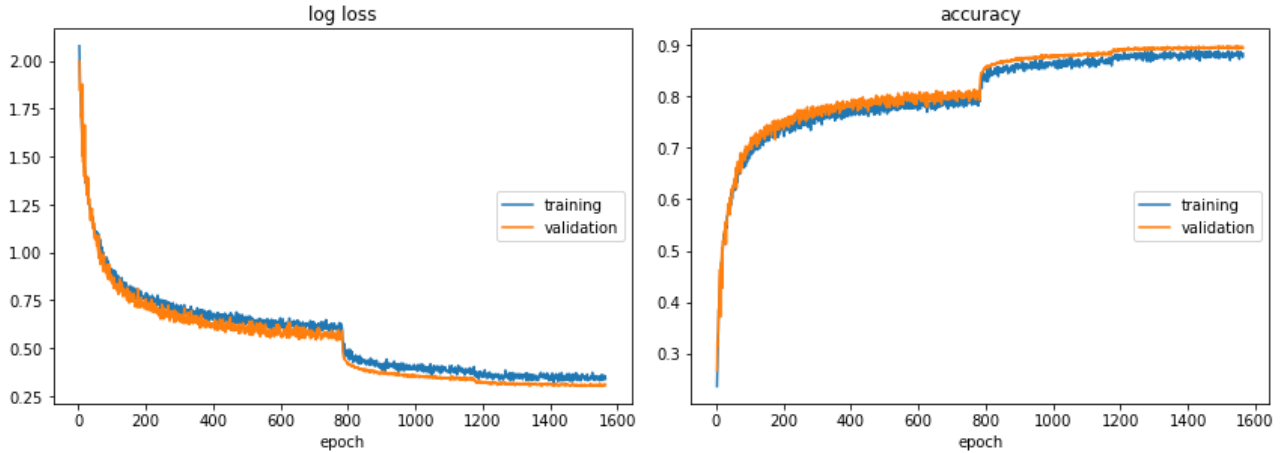


Figure 6.3. HBONext v1.2 (a) Accuracy and Losses vs Epochs (Augmentation)

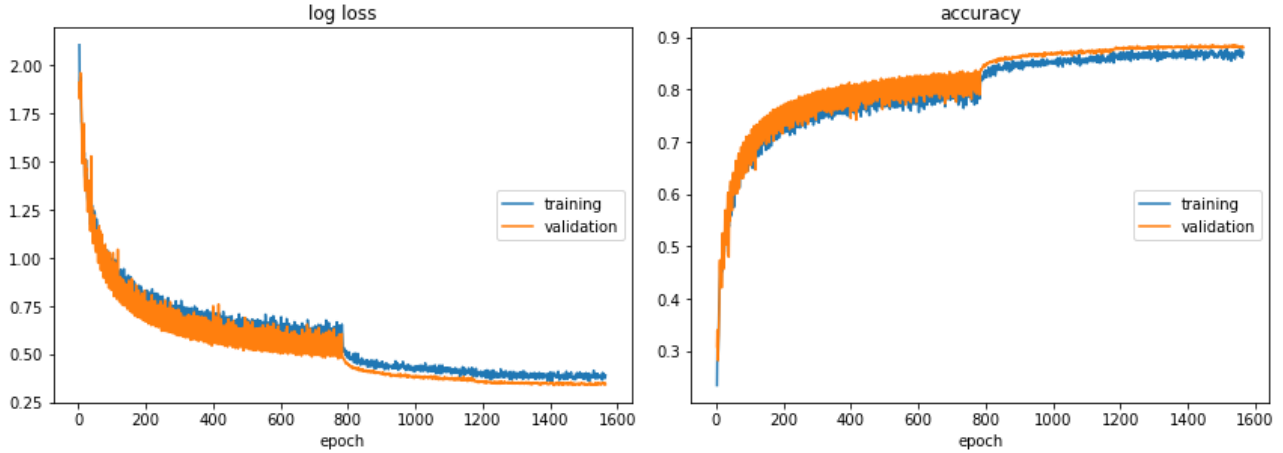


Figure 6.4. HBONext v1.2 (b) Accuracy and Losses vs Epochs (Augmentation + L_r Scheduling)

6.3 Model Size Variants

The training graphs on CIFAR-10 are visualized using the Pytorch platform and the Livelossplot software, as seen in Figure 6.1, which is a plot of Accuracy Performance to the number of epochs obtained for a width value of 1.0. The proposed model HBONext v1.1 has an accuracy of 88.30% with a model size of 7.66 MB, which is better compared to the baseline model’s accuracy of 80.97% and initial model size of 22 M. The model versions are summarized in Table 6.3 based on their width multiplier values. This model was successfully trained on CIFAR-10 with a particular width multiplier value using the Google Colab environment. To spot the variations, the corresponding precision and model size values are carefully recorded.

Table 6.3. Width Multiplier Variants of HBONext v1.1

Width Multiplier	Accuracy	Model size
HBONext(1.5)	89.60%	16.08 MB
HBONet (1.5)	82.75%	48.34 MB
HBONext(1.0)	88.30%	7.66 MB
HBONet (1.0)	80.97%	22.00 MB
HBONext(0.75)	87.70%	4.67 MB
HBONet (0.75)	79.93%	13.80 MB
HBONext(0.50)	85.30%	2.48 MB
HBONet (0.50)	76.25%	7.04 MB
HBONext(0.25)	79.80%	1.07 MB
HBONet (0.25)	71.22%	2.65 MB

6.4 Hardware Validation

6.4.1 NXP's BLBX2 Implementation Steps

We would be able to build and implement image classifier algorithms for autonomous applications such as image recognition, object detection, and more using the Python portion of RTMaps linked to the desktop.

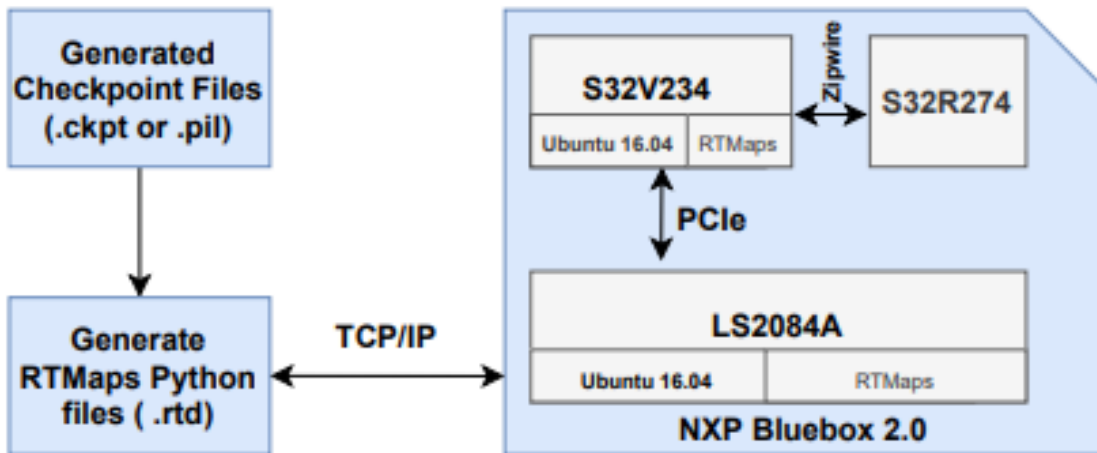


Figure 6.5. Interfacing Overview with NXP BLBX2

The python section includes an editor that assists users in writing, compiling, and deploying python scripts. To execute users' python scripts within embedded hardware, this

editor has three main functions that must be understood. The three functions available in the editor are:

- Birth() to start the method.
- Core() to run it.
- Death() to stop it.

In order to initiate and set the python code, Birth() is named at the beginning of the program. Core() is an algorithm that runs forever. As a result, the user’s code can be placed in this part of the code and run forever. When a program is ended, the operation Death() is called, and it is declared at the end of the program [39].

We can use RTMaps to run the program on the hardware Bluebox 2.0 through NXP after the python script is written and finished, which helps with execution. Figure 6.5, illustrates how RTMaps are configured with Bluebox 2.0. The host machine and the Bluebox, which serves as the target hardware, communicate using the TCP/IP protocol. After connecting to the host computer, the user can search the right communication ports in the device manager. The user can then configure Tera term/Putty for the LS2 and S32V interfaces. We use GPU to train the model and save learned checkpoints, which we then merge with our running Python script on RTMaps and validate on Bluebox 2.0. The proposed HBONext architecture classifier’s deployment procedure on the Bluebox2.0 application platform as seen above in our publication on HBONext deployment. The execution engine executes the software on the Linux Operating System of our Bluebox2.0, and RTMaps studio creates a TCP/IP connection with it. It includes a Python block for successfully building and deploying PyTorch framework-based code on the machine.

6.4.2 NXPs BLBX2 Implementation Results

Table 6.4, describes the key parameters taken into consideration when implementing HBONext architecture on NXP’s Bluebox 2.0 edge hardware. After initial testing locally, the suggested HBONext classification technique is tested on the RTMaps Studio platform and then introduced on the NXP Bluebox 2.0. Figure 6.6, shows the RTMaps version connected

to the RTMaps Studio on the computer, which uses remote engine networking controlled by TCP/IP to provide debug functionality and a graphical interface.

Table 6.4. Key Implementation Parameters of HBONext v1.2

Width Multiplier	Accuracy	Model size
Proposed HBONext v1.2 (a) (1.5)	89.70%	3.00 MB

This Figure 6.6, shows a general description of the RTMaps Console. Before sending the script to the Bluebox SD card, it is checked on RTMaps attached to a local device. We also use RTMaps to construct an image classifier instance that chooses between the baseline HBONet model and our proposed HBONext model before making predictions based on the class label. In Figure 6.7, we fed this classifier an image of an aircraft, and it correctly predicted the class.

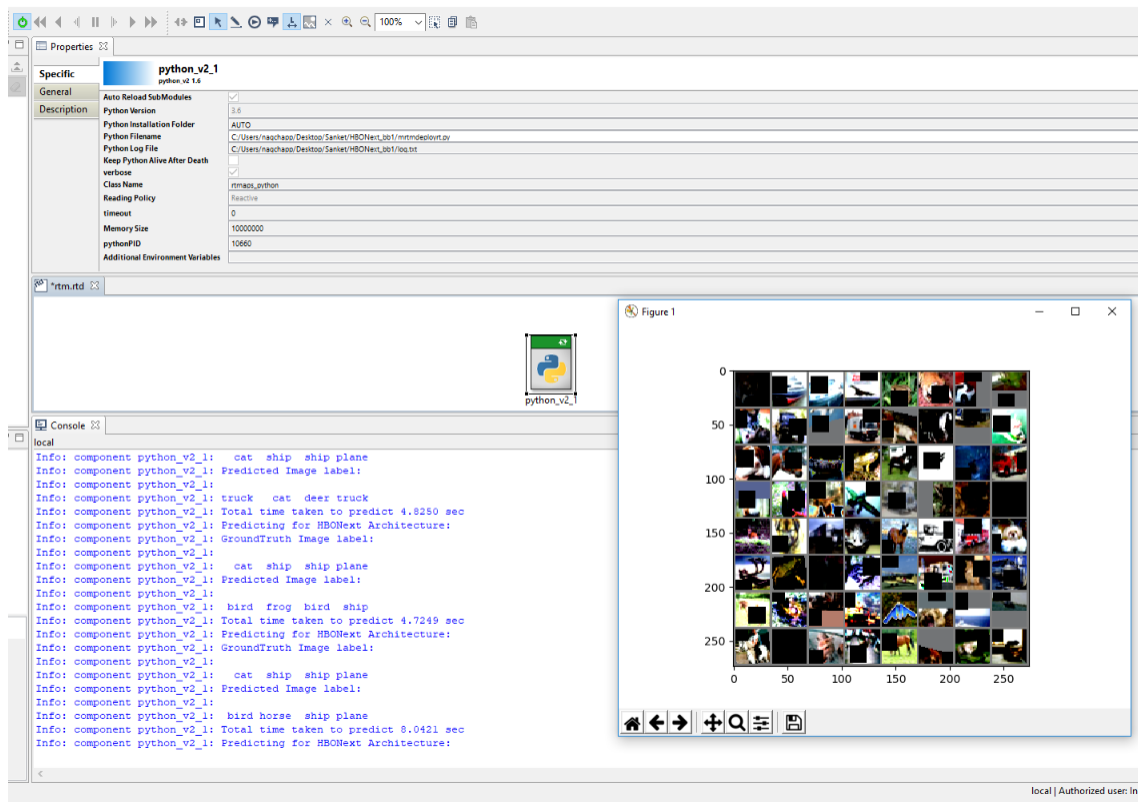


Figure 6.6. HBONext Architecture Testing on RTMaps

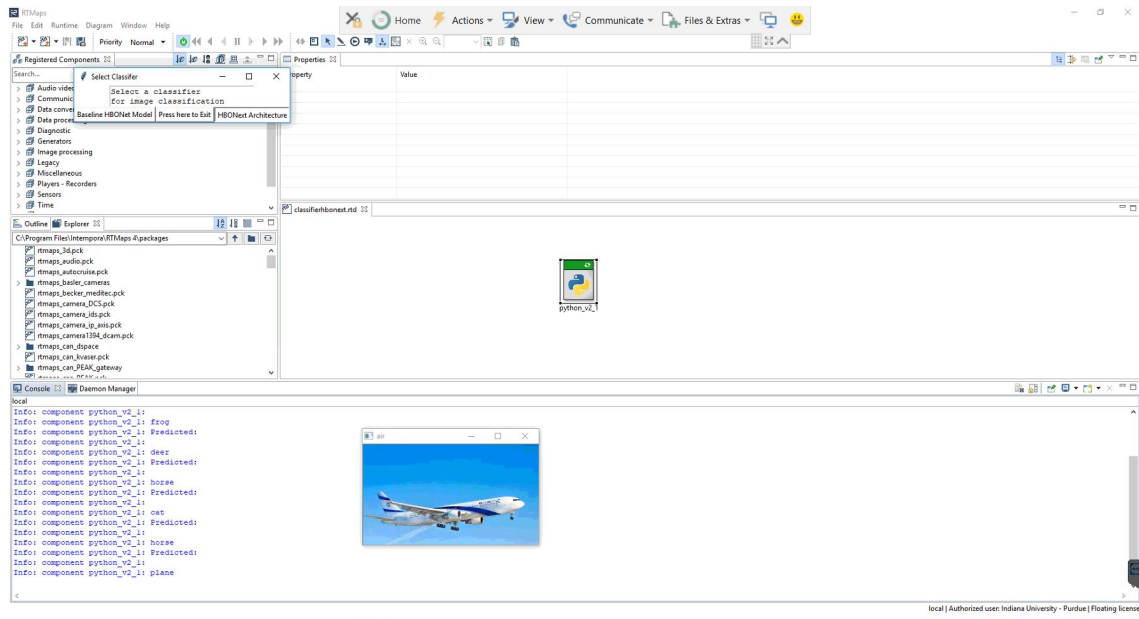


Figure 6.7. Image Classifier using HBONext v1.2 on RTMaps

Also, the Figure 6.8, shows the image classification for CIFAR-10 using HBONext v1.2 (using Table 6.4) on embedded hardware BlueBox 2.0 by NXP.

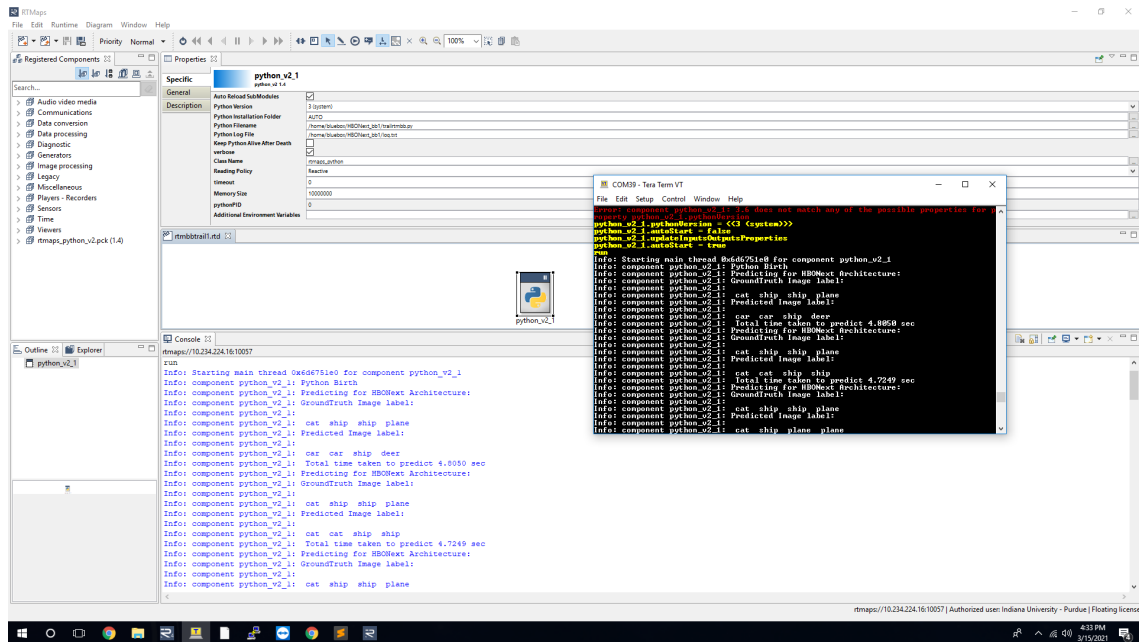
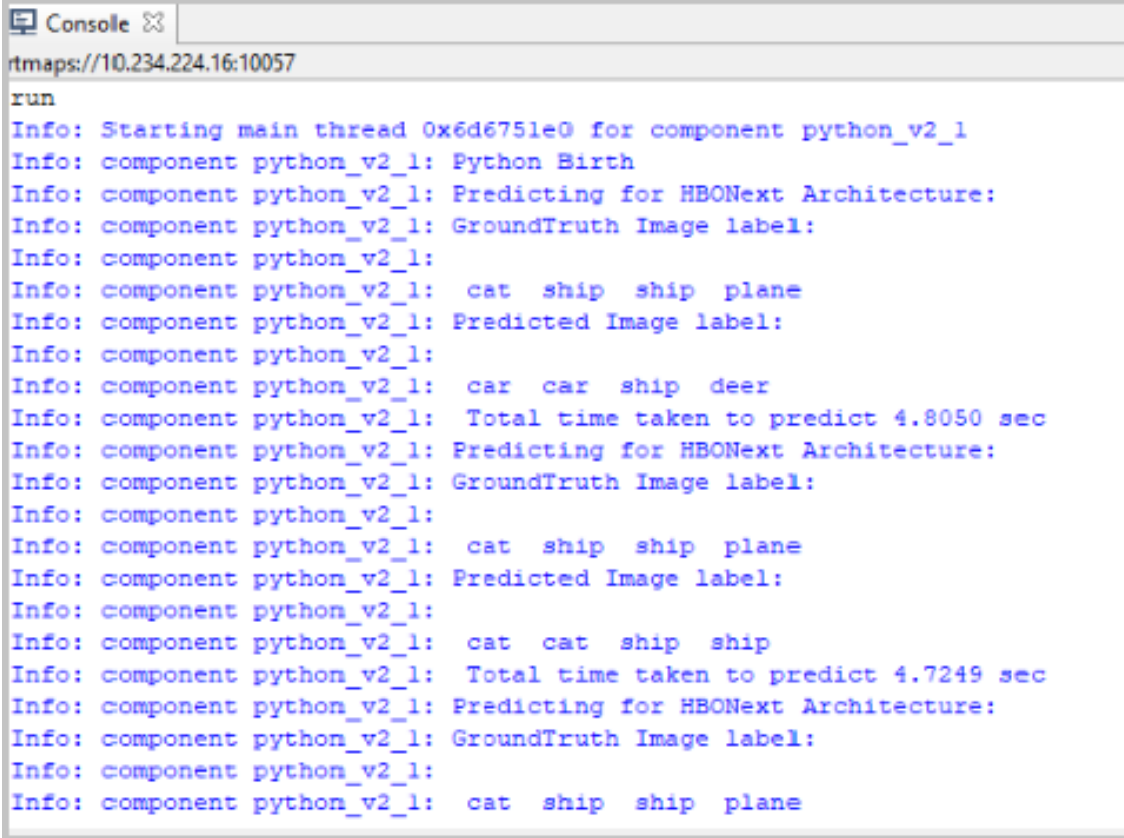


Figure 6.8. Image Classification for CIFAR-10 using HBONext v1.2

Testing Results on RTMaps Console:

The RTMaps, as seen in the same diagram, is made up of the python module, which enables python-based code to be integrated with RTMaps. Figure 6.9, shows the output of the testing results in a console view. In the classification approach, the model learns the mapping between the input and the output attribute, which is a label.



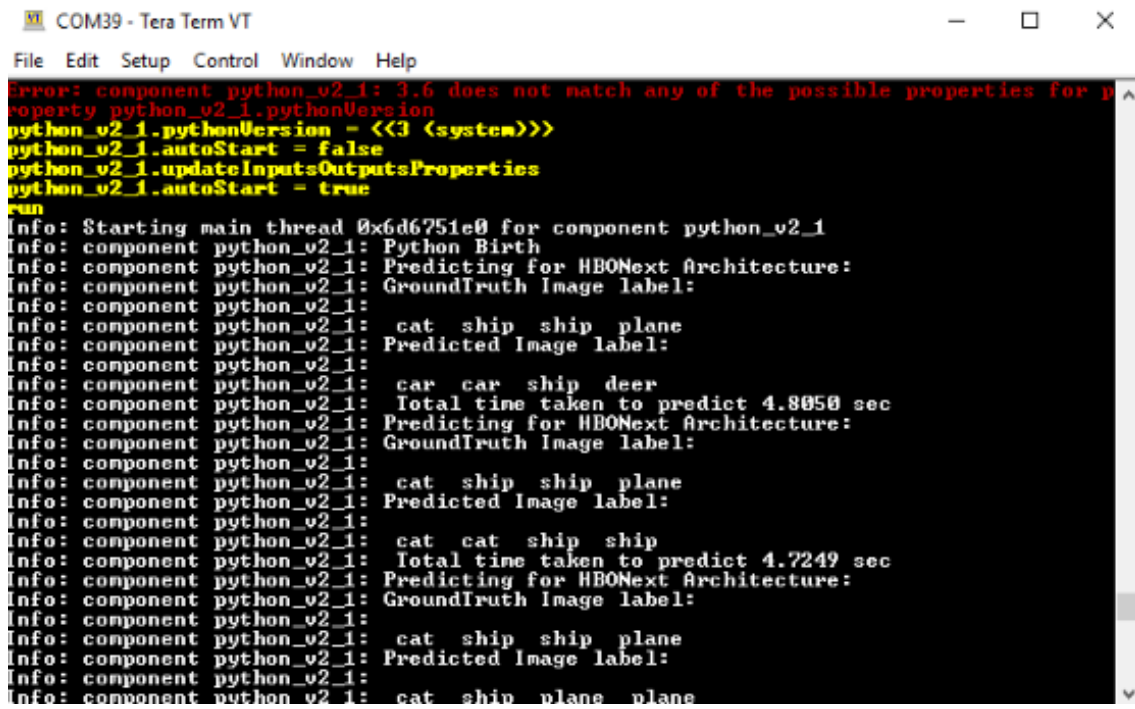
```
rtmaps://10.234.224.16:10057
run
Info: Starting main thread 0x6d6751e0 for component python_v2_1
Info: component python_v2_1: Python Birth
Info: component python_v2_1: Predicting for HBONext Architecture:
Info: component python_v2_1: GroundTruth Image label:
Info: component python_v2_1:
Info: component python_v2_1: cat ship ship plane
Info: component python_v2_1: Predicted Image label:
Info: component python_v2_1:
Info: component python_v2_1: car car ship deer
Info: component python_v2_1: Total time taken to predict 4.8050 sec
Info: component python_v2_1: Predicting for HBONext Architecture:
Info: component python_v2_1: GroundTruth Image label:
Info: component python_v2_1:
Info: component python_v2_1: cat ship ship plane
Info: component python_v2_1: Predicted Image label:
Info: component python_v2_1:
Info: component python_v2_1: cat cat ship ship
Info: component python_v2_1: Total time taken to predict 4.7249 sec
Info: component python_v2_1: Predicting for HBONext Architecture:
Info: component python_v2_1: GroundTruth Image label:
Info: component python_v2_1:
Info: component python_v2_1: cat ship ship plane
```

Figure 6.9. RTMaps Console Output

Validation on TeraTerm Window:

The proposed architecture was trained with the CIFAR-10 dataset, and the checkpoints files were stored during the training phase so that they could be loaded later with the RTMaps Python package. The Core() part is written with the model checkpoint files and later a few random pictures are transferred from the test dataset folder with the right ground truth picture labels, and the model is asked to predict these random picture labels for validation

with a few lines of enabling code. The Teraterm terminal is used to further interpret the results on Bluebox 2.0 and check them with the RTMaps console, as seen in



```
COM39 - Tera Term VT
File Edit Setup Control Window Help
error: component python_v2_1: 3.6 does not match any of the possible properties for p
property python_v2_1.pythonVersion
python_v2_1.pythonVersion = <<3 <system>>>
python_v2_1.autoStart = false
python_v2_1.updateInputsOutputsProperties
python_v2_1.autoStart = true
run
Info: Starting main thread 0x6d6751e0 for component python_v2_1
Info: component python_v2_1: Python Birth
Info: component python_v2_1: Predicting for HBONext Architecture:
Info: component python_v2_1: GroundTruth Image label:
Info: component python_v2_1: cat ship ship plane
Info: component python_v2_1: Predicted Image label:
Info: component python_v2_1:
Info: component python_v2_1: car car ship deer
Info: component python_v2_1: Total time taken to predict 4.8050 sec
Info: component python_v2_1: Predicting for HBONext Architecture:
Info: component python_v2_1: GroundTruth Image label:
Info: component python_v2_1: cat ship ship plane
Info: component python_v2_1: Predicted Image label:
Info: component python_v2_1:
Info: component python_v2_1: cat cat ship ship
Info: component python_v2_1: Total time taken to predict 4.7249 sec
Info: component python_v2_1: Predicting for HBONext Architecture:
Info: component python_v2_1: GroundTruth Image label:
Info: component python_v2_1:
Info: component python_v2_1: cat ship ship plane
Info: component python_v2_1: Predicted Image label:
Info: component python_v2_1:
Info: component python_v2_1: cat ship plane plane
```

Figure 6.10. TeraTerm Console Validation Results

7. CONCLUSIONS AND FUTURE WORK

7.1 Conclusion

The work highlights, HBONext block a mixture of derived HBO and a Flipped variant of the Inverted Residual (FIR) block. HBO makes use of interdependencies across the spatial and channel parameters of depth-wise convolution in a bilateral symmetric manner, first with spatial contraction-expansion and then with a channel expansion-contraction component. In comparison to traditional inverted residual blocks, FIR relies on higher-dimensional aspect details and stresses the use of an extra spatial convolution. Thus, this work takes advantage of the benefits of these two approaches to build a lightweight CNN based on the principle of depth-wise separable convolution with incredibly restricted computing memory resources, as well as an easy to deploy version of this model for any embedded edge hardware. Using the CIFAR-10 dataset and the proposed HBONext architecture, this study illustrates image classification competency. The research compares our model to various width multiplier values trained with optimization techniques and cosine annealing scheduling methods for learning. Lighter versions are accomplished by adjusting the width multipliers' values, which can be effectively applied on any embedded vision program. The proposed model will be deployed on embedded edge hardware in the future to validate its real-time application for image classification. The CIFAR-10 dataset is successfully used for a real-time image classification purpose on a very powerful and scalable embedded hardware NXP Blubox 2.0. The model is just 3 MB in size and has an accuracy of 89.70%, making it a perfect choice for any vision-based embedded platform. There are also ways to shrink the model even further, as well as several techniques for increasing model accuracy, which will be addressed further. This model can also be used to better understand object detection and tracking capability.

7.2 Future Work

1. It can also be used to create a model that can detect and localize objects in an image for object detection. We also plan to focus on demonstrating these two versions of the model using the SSD (Single Shot Multi-Box Detector).

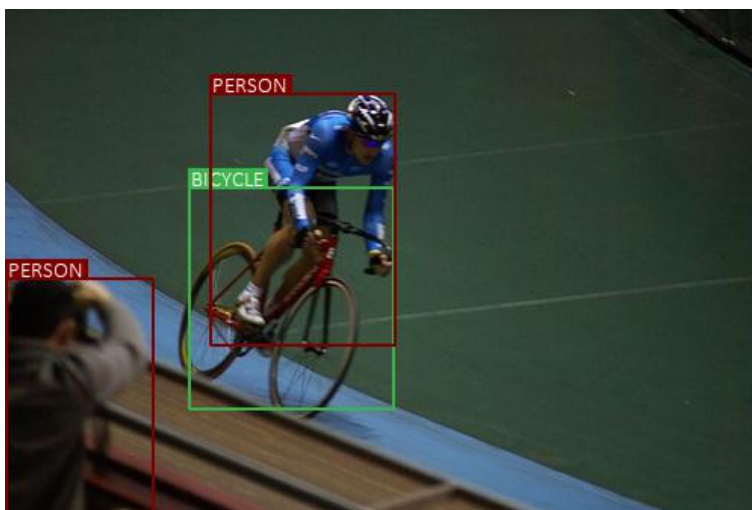


Figure 7.1. Example for Object Detection

2. Deep CNN (DCNN) approaches are very non-linear, and they often provide improved versatility based on various enhancement strategies or scaling the volume of training data to these methods. The only drawback of these models is that they are highly sensitive to the specific details of train data, and they may learn new weights each time, resulting in different predictions. An ensemble learning technique can be used to solve the high variance of these networks. This technique involves training initial weights on the same data using various networks of identical configurations. Each model makes a prediction, and the final prediction is the sum of all the predictions produced by the various models.
3. Deep CNN (DCNN) methods are often challenged by concerns such as computation complexity, memory constraints, and the number of parameters used in each learning process. Our work has resulted in a smaller model size for embedded

edge applications, but ‘model pruning’ is a well-known technique for developing smaller, simpler, and more effective models.

4. Further, larger datasets such as CIFAR-100, ImageNet, SVHN, and others can be used to train the proposed HBONext architecture.

REFERENCES

- [1] W. Rawat and Z. Wang, “Deep convolutional neural networks for image classification: A comprehensive review,” *Neural computation*, vol. 29, no. 9, pp. 2352–2449, 2017.
- [2] X. Wang, Y. Zhao, and F. Pourpanah, “Recent advances in deep learning,” *International Journal of Machine Learning and Cybernetics*, vol. 11, no. 4, pp. 747–750, 2020. DOI: [10.1007/s13042-020-01096-5](https://doi.org/10.1007/s13042-020-01096-5).
- [3] I. Goodfellow, Y. Bengio, A. Courville, and Y. Bengio, *Deep learning*, 2. MIT press Cambridge, 2016, vol. 1.
- [4] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “Imagenet classification with deep convolutional neural networks,” *Advances in neural information processing systems*, vol. 25, pp. 1097–1105, 2012.
- [5] Y. Lecun, Y. Bengio, and G. Hinton, “Deep learning,” *Nature*, vol. 521, no. 7553, pp. 436–444, 2015, ISSN: 0028-0836. DOI: [10.1038/nature14539](https://doi.org/10.1038/nature14539). [Online]. Available: <https://dx.doi.org/10.1038/nature14539>.
- [6] A. Geiger, P. Lenz, C. Stiller, and R. Urtasun, “Vision meets robotics: The kitti dataset,” *The International Journal of Robotics Research*, vol. 32, no. 11, pp. 1231–1237, 2013.
- [7] É. Zablocki, H. Ben-Younes, P. Pérez, and M. Cord, “Explainability of vision-based autonomous driving systems: Review and challenges,” *arXiv preprint arXiv:2101.05307*, 2021.
- [8] M. Xie, L. Trassoudaine, J. Alizon, M. Thonnat, and J. Gallice, “Active and intelligent sensing of road obstacles: Application to the european eureka-prometheus project,” in *1993 (4th) International Conference on Computer Vision*, IEEE, 1993, pp. 616–623.
- [9] G. W. Lindsay, “Convolutional neural networks as a model of the visual system: Past, present, and future,” *Journal of cognitive neuroscience*, pp. 1–15, 2020.
- [10] Y. Bengio, T. Mesnard, A. Fischer, S. Zhang, and Y. Wu, “Stdp-compatible approximation of backpropagation in an energy-based model,” *Neural computation*, vol. 29, no. 3, pp. 555–577, 2017.
- [11] D. Sirohi, N. Kumar, and P. S. Rana, “Convolutional neural networks for 5g-enabled intelligent transportation system: A systematic review,” *Computer Communications*, vol. 153, pp. 459–498, 2020.
- [12] W. S. McCulloch and W. Pitts, “A logical calculus of the ideas immanent in nervous activity,” *The bulletin of mathematical biophysics*, vol. 5, no. 4, pp. 115–133, 1943.

- [13] D. H. Hubel and T. N. Wiesel, “8. receptive fields of single neurones in the cat’s striate cortex,” in *Brain Physiology and Psychology*, University of California Press, 2020, pp. 129–150.
- [14] D. H. Hubel and T. N. Wiesel, “Receptive fields, binocular interaction and functional architecture in the cat’s visual cortex,” *The Journal of physiology*, vol. 160, no. 1, pp. 106–154, 1962.
- [15] K. Fukushima, “Self-organization of a neural network which gives position-invariant response,” in *Proceedings of the 6th international joint conference on Artificial intelligence-Volume 1*, 1979, pp. 291–293.
- [16] P. Werbos, “Beyond Regression: New tools for Prediction and Analysis in the Behavioral Sciences,” *Ph. D. dissertation, Harvard University*, 1974.
- [17] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, “Learning representations by back-propagating errors,” *Nature*, vol. 323, no. 6088, pp. 533–536, 1986, ISSN: 0028-0836. DOI: [10.1038/323533a0](https://dx.doi.org/10.1038/323533a0). [Online]. Available: <https://dx.doi.org/10.1038/323533a0>.
- [18] Y. LeCun, Y. Bengio, *et al.*, “Convolutional networks for images, speech, and time series,” *The handbook of brain theory and neural networks*, vol. 3361, no. 10, p. 1995, 1995.
- [19] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, “Gradient-based learning applied to document recognition,” *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998. DOI: [10.1109/5.726791](https://doi.org/10.1109/5.726791).
- [20] D. DeCoste and B. Schölkopf, “Training invariant support vector machines,” *Machine learning*, vol. 46, no. 1, pp. 161–190, 2002.
- [21] R. Collobert, F. Sinz, J. Weston, L. Bottou, and T. Joachims, “Large scale transductive svms.,” *Journal of Machine Learning Research*, vol. 7, no. 8, 2006.
- [22] M. R. Minar and J. Naher, “Recent advances in deep learning: An overview,” *arXiv preprint arXiv:1807.08169*, 2018.
- [23] C. Lemaréchal, “Cauchy and the gradient method,” *Doc Math Extra*, vol. 251, no. 254, p. 10, 2012.
- [24] Z. J. Wang, R. Turko, O. Shaikh, H. Park, N. Das, F. Hohman, M. Kahng, and D. H. Chau, “Cnn explainer: Learning convolutional neural networks with interactive visualization,” *IEEE Transactions on Visualization and Computer Graphics*, 2020.
- [25] D. Li, A. Zhou, and A. Yao, “Hbonet: Harmonious bottleneck on two orthogonal dimensions,” in *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2019, pp. 3316–3325.
- [26] Z. Daquan, Q. Hou, Y. Chen, J. Feng, and S. Yan, “Rethinking bottleneck structure for efficient mobile network design,” *arXiv preprint arXiv:2007.02269*, 2020.

- [27] S. Venkitachalam, S. K. Manghat, A. S. Gaikwad, N. Ravi, S. B. S. Bhamidi, and M. El-Sharkawy, “Realtime applications with rtmmaps and bluebox 2.0,” in *Proceedings on the International Conference on Artificial Intelligence (ICAI)*, The Steering Committee of The World Congress in Computer Science, Computer ..., 2018, pp. 137–140.
- [28] D. Katare and M. El-Sharkawy, “Embedded system enabled vehicle collision detection: An ann classifier,” in *2019 IEEE 9th Annual Computing and Communication Workshop and Conference (CCWC)*, IEEE, 2019, pp. 0284–0289.
- [29] N. Semiconductors, *S32v234, s32v234 data sheet*, <https://www.nxp.com/docs/en/data-sheet/S32V234.pdf>, (Accessed on 04/21/2021), Mar. 2020.
- [30] N. Semiconductors, *QorIQ ls2084a/ls2044a data sheet*, <https://www.nxp.com/docs/en/data-sheet/LS2084A.pdf>, (Accessed on 04/21/2021), Sep. 2020.
- [31] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, *et al.*, “Tensorflow: Large-scale machine learning on heterogeneous distributed systems,” *arXiv preprint arXiv:1603.04467*, 2016.
- [32] A. Gulli and S. Pal, *Deep learning with Keras*. Packt Publishing Ltd, 2017.
- [33] PyTorch, *Github - pytorch/pytorch: Tensors and dynamic neural networks in python with strong gpu acceleration*, <https://github.com/pytorch/pytorch>, (Accessed on 04/21/2021).
- [34] I. Kostrikov, *Pytorch implementations of reinforcement learning algorithms*, <https://github.com/ikostrikov/pytorch-a2c-ppo-acktr-gail>, 2018.
- [35] K. O. Stanley and R. Miikkulainen, “Evolving neural networks through augmenting topologies,” *Evolutionary computation*, vol. 10, no. 2, pp. 99–127, 2002.
- [36] T. Contributors, *Elu — pytorch 1.8.1 documentation*, <https://pytorch.org/docs/stable/generated/torch.nn.ELU.html>, (Accessed on 04/21/2021), 2019.
- [37] D. Misra, “Mish: A self regularized non-monotonic activation function,” *arXiv preprint arXiv:1908.08681*, 2019.
- [38] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, “Mobilenets: Efficient convolutional neural networks for mobile vision applications,” *arXiv preprint arXiv:1704.04861*, 2017.
- [39] M. Ayi and M. El-Sharkawy, “Real-time implementation of rnmv2 classifier in nxp bluebox 2.0 and nxp i. mx rt1060,” in *2020 IEEE Midwest Industry Conference (MIC)*, IEEE, vol. 1, 2020, pp. 1–4.

PUBLICATIONS

- S. R. Joshi and M. El-Sharkawy, “HBONext: HBONet with Flipped Inverted Residual”, in 3rd IEEE 2021 International Conference on Design Test of integrated micro nano-Systems, 2021. (Accepted)
- S. R. Joshi and M. El-Sharkawy, “Hardware Deployment of HBONext using NXP Bluebox 2.0”, in IEEE World AI IoT Congress, 2021. (Accepted)