DESIGN SPACE EXPLORATION OF MOBILENET FOR SUITABLE HARDWARE DEPLOYMENT

A Thesis

Submitted to the Faculty

of

Purdue University

by

Debjyoti Sinha

In Partial Fulfillment of the

Requirements for the Degree

of

Master of Science in Electrical and Computer Engineering

May 2020

Purdue University

Indianapolis, Indiana

THE PURDUE UNIVERSITY GRADUATE SCHOOL

STATEMENT OF THESIS APPROVAL

This is dedicated to my beloved mother Shampa Sinha, father Uday Kumar Sinha,
to all my dear friends and IoT Collaboratory colleagues

ACKNOWLEDGMENTS

I would like to thank my adviser Dr. Mohamed El Sharkawy for his constant guidance, motivation and support and providing all the resources needed for the completion of this research. I would also like to extend my gratitude towards other committee members Dr. Brian King and Dr. Maher Rizkalla for their valuable inputs.

I would also like to thank Sherrie Tucker, the Electrical and Computer Engineering Department coordinator for her timely assistance in finishing up my work before deadlines.

Finally, I would like to thank all my friends in the IoT Collaboratory lab for supporting me and maintaining a good working environment. Among my friends, I would like to give a special thanks to Saurabh Ravindra Desai for assisting me with the deployment of my model into the i.MX RT1060 hardware.

TABLE OF CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

ABBREVIATIONS

Adam	Adaptive Moment Estimation
ADAS	Advanced Driver Assistance Systems
AI	Artificial Intelligence
API	Application Programming Interface
Avg	Average
BLBX2.0	Bluebox2.0
BN	Batch Normalization
BSP	Board Support Package
CNN	Convolutional Neural Network
CSI	Camera Serial Interface
CV	Computer Vision
DSI	Display Serial Interface
DL	Deep Learning
DNN	Deep Neural Network
DRAM	Dynamic Random Access Memory
DSE	Design Space Exploration
ECSPI	Enhanced Configurable Serial Peripheral Interface
e.g.	Example
ELU	Exponential Linear Unit
FC	Fully Connected
GB	Gigabyte
ILSVRC	ImageNet Large Scale
IoT	Internet of Things
KB	Kilobyte

LCD	Liquid Crystal Display
LR	Learning Rate
Max	Maximum
MCU	Microcontroller unit
ML	Machine Learning
Nadam	Nesterov Adaptive Moment Estimation
NN	Neural Network
OTG	On The Go
PCI	Peripheral Component Interconnect
PDM	Pulse Density Modulation
RAM	Random Access Memory
ReLU	Rectified Linear Unit
ROM	Read Only Memory
RNN	Recurrent Neural Network
SDK	Software Development Kit
SELU	Scaled Exponential Linear Unit
SIMD	Single Instruction Multiple Data
SGD	Stochastic Gradient Descent
SPI	Serial Peripheral Interface
TF	Tensor Flow
UART	Universal Asynchronous Receiver Transmitter
UAV	Unmanned Aerial Vehicle
USB	Universal Serial Bus
VPU	Video Processing Unit
VRC	Visual Recognition Competition

ABSTRACT

Sinha, Debjyoti. M.S.E.C.E., Purdue University, May 2020. Design Space Exploration of MobileNet For Suitable Hardware Deployment. Major Professor: Mohamed El-Sharkawy.

Designing self-regulating machines that can see and comprehend various real world objects around it are the main purpose of the AI domain. Recently, there has been marked advancements in the field of deep learning to create state-of-the-art DNNs for various CV applications. It is challenging to deploy these DNNs into resource-constrained micro-controller units as often they are quite memory intensive. Design Space Exploration is a technique which makes CNN/DNN memory efficient and more flexible to be deployed into resource-constrained hardware. MobileNet is small DNN architecture which was designed for embedded and mobile vision, but still researchers faced many challenges in deploying this model into resource limited real-time processors.

This thesis, proposes three new DNN architectures, which are developed using the Design Space Exploration technique. The state-of-the art MobileNet baseline architecture is used as foundation to propose these DNN architectures in this study. They are enhanced versions of the baseline MobileNet architecture. DSE techniques like data augmentation, architecture tuning, and architecture modification have been done to improve the baseline architecture. First, the Thin MobileNet architecture is proposed which uses more intricate block modules as compared to the baseline MobileNet. It is a compact, efficient and flexible architecture with good model accuracy. To get a more compact models, the KilobyteNet and the Ultra-thin MobileNet DNN architecture is proposed. Interesting techniques like channel depth alteration and hyperparameter tuning are introduced along-with some of the techniques used for designing the Thin MobileNet. All the models are trained and validated from scratch on the CIFAR-10 dataset. The experimental results (training and testing) can be visualized using the live accuracy and logloss graphs provided by the Liveloss package. The Ultra-thin MobileNet model is more balanced in terms of the model accuracy and model size out of the three and hence it is deployed into the NXP i.MX RT1060 embedded hardware unit for image classification application.

1. INTRODUCTION

Artificial Intelligence (AI) is the engineering which develops intelligent machines and programs. Machine Learning is a subset of AI which is mainly concerned with the ability of machines to learn data without being explicitly programmed. Deep learning is a subdomain of Machine Learning in AI that has networks called Deep Neural Networks that are capable of unsupervised learning from unstructured data. This is achieved through various learning algorithms and optimization techniques. Deep Neural Networks (DNN) gained popularity when AlexNet [1] won the ImageNet Challenge in the year 2012. Since then, the domain of deep learning expanded exponentially. Many standard algorithms for computer vision like Canny algorithm or HOG have been replaced by deep learning models like SqueezeNet [2], SqueezeNext [3], ResNet [4], Inception [5], etc. There also had been significant developments in new optimization techniques, non-linear activation functions, training methods, etc. In order to get higher accuracies, the models are made deeper and complex. The advent of deeper and complicated models has led to the development of a wide number of hardware architectures like GPUs, Bluebox 2.0, S32V234 MCU, etc to increase the speed of the training process and deploy the models for various computer vision applications. But increasing the depth and complexity of a model increases the size and computation cost making it less efficient for hardware deployment, especially in resource-constrained mobile and embedded platforms.

In real-time applications such as image classification, image captioning [6] object detection [7] and semantic segmentation [8,9] in autonomous driving [10], the inference time and accuracy are very important factors to safety. So, it becomes necessary to have a model which is very accurate, requires less memory complexity and has less computation time for its utilization in real-time scenarios. DNNs came out of existing algorithms like HOG and SIFT and they have performed well in image classification, object recognition, detection and segmentation tasks. RNN have been successful in generating text from visual stimulus. DNN are more accurate when it comes to computer vision applications, but the issue with them is that they are computationally intensive and memory consuming.

1.1 Context

DNNs have become powerful tools as for CV applications in various industries. DSE of CNN, new strategies and techniques have led to the development of memory and computationally efficient DNN models. This has made CNN suitable to be deployed into resource-constrained embedded platforms. The main idea of this research work is to propose efficient or adaptable deep neural network models which are compact, less computationally intensive and having competitive accuracy levels at the same time.

The expanding intricacy of NN has further prompted the use of parallel-distributed architectures and complex hardware units for computation like the S32V234, BLBX2.0, NVIDIA TITAN, GTX 1080, TESLA GPUs, etc. These units accelerate the training process of different DNN. Apart from the advanced hardware units, DL has made significant progress in CV due to the advent of advanced software frameworks like TF, Keras, PyTorch, Theano, Caffe, etc. There is also a large open source community associated with this particular domain to help researchers and academicians develop more efficient DNNs, by improving these software frameworks time to time. This further encouraged a larger group of people to acquire the knowledge, skill and expertise, and train and test CNNs rapidly. Datasets like the ImageNet, CIFAR-10, CIFAR-100 are used for the training of deep architectures. Transfer learning has made numerous pre-trained models readily available. These pre-trained architectures have shown good results during the model testing phase. Even though the state-of-the-art GPUs and advanced software frameworks have made the training of DNN and image classification tasks easier, there is an increasing need to embed these applications into real-time processors with power, size and memory constraints. Deeper neural network architectures need more resources than shallow networks. For obtaining more accurate DNNs, researchers have concentrated more on increasing the depth and complexity of the DNNs. The shallow DNN architectures, on the other hand, concentrated only on the model size and not on the model speed. There is a need to make the models more compact, keeping the accuracy level same as those of the CNNs, with a good model speed at the same time, so that they can be easily deployed easily into resource-constrained processors and can be reliably utilized for real-time autonomous applications. The inference time should be in milliseconds when they are used for real-time computer vision applications, otherwise there will always be a question mark on the reliability of these DNNs in real-time scenarios.

There are some exemplary DNN models which have achieved good competitive accuracy, small model size and good model speed such as SqueezeNet, SqueezeNext and MobileNet [11]. This research work proposes two similar DNN architectures which would be easily deployable into resource-constrained autonomous hardware platforms for real-time CV applications.

1.2 Motivation

Image classification, object detection, object recognition are challenging tasks for a machine. Various CV algorithms are the basis to develop applications for autonomous cars, surveillance systems, drones, UAVs, etc. As the quality of self-governance of a system increases, the need to design an algorithm with intuitions rises. Significant developments in the area of CV algorithms have been made in the past few years, and hence it gave rise to more sophisticated CV systems. Machine Learning is a novel AI methodology, where the hard-coded attributes of an image are not searched for, rather a machine is trained to learn the image attributes with the help of DNNs. This can be thought as the brain of a human child learning to identify various objects. The training and testing of DNNs has become much easier due to the advent of advanced hardware and software platforms.

As mentioned earlier, deployment of DNNs becomes a challenge when the size of the model is in the range between few Megabytes to Gigabytes, as it leads to a big memory overhead. Big models also take a good amount of time to execute CV tasks. The use of GPUs can be a solution but they are not compact enough to be used within the target devices right now. This thesis explores the DSE technique to make a DNN model compact and have less computation time with minimum trade off with the model accuracy.

1.3 Problem Statement

This thesis explores the DL field by dealing with the following:

1. Importance and effect of the Design Space Exploration method.
2. Effect of architectural tuning on the baseline DNN.
3. Effect of architectural modifications on the baseline DNN.
4. Monitoring the DNN accuracy and losses after every epoch.
5. Deployment into NXP i.MX RT1060 hardware

1.4 Roadblocks

1. Training and testing of DNN at good pace.
2. Appropriate DSE tools for the development of the new DNN architectures.
3. Maintaining a competitive model accuracy alongwith small model size.
4. Solving the overfitting problem which is present in the baseline DNN.
5. Configuring the autonomous hardware for deploying the model into it.

1.5 Contribution

This thesis primarily focuses on the different DSE techniques induced on a baseline DNN to develop three new DNN architectures which are more efficient than the baseline version. The foundation architecture is the baseline MobileNet v1 DNN [11] and the proposed frameworks are: Thin MobileNet, KilobyteNet and Ultra-thin MobileNet. The proposed models are better than the baseline MobileNet v1 model in some aspects like the mode accuracy or the model size. A competitive model accuracy is maintained in case of the proposed architectures and the accuracy levels are higher than the baseline accuracy level for two of the proposed architectures. A significant reduction in the total number of parameters, and hence the model size, is achieved. The Thin MobileNet DNN contains only 25% of the total number of parameters contained in the baseline MobileNet v1. The average computation time is also about 17s less than the baseline version. The KilobyteNet is an enhanced version of the MobileNet with a model size in the kilobyte range and better model speed than the MobileNet v1. The Ultra-thin MobileNet is another enhanced version of the MobileNet like the Thin MobileNet and contains only 9.37% of the total number of parameters contained in the baseline MobileNet v1. The average computation time of this architecture is about 15s less than the baseline MobileNet. There is almost negligible overfitting problem in these two models. Apart from comparing the proposed models with the baseline model, a comparison with other benchmark architectures is also drawn. These models are trained and tested on the CIFAR-10 dataset and the finally deployed into the autonomous embedded processor, NXP i.MX RT1060 using the MCUXpresso SDK respectively for image classification application, both with and without camera.

2. LITERATURE REVIEW

AI has narrowed down the difference between humans and machines to some extent. Researchers in the field of CV have been trying to develop more advanced algorithms and architectures using the analogy of the human brain and machines.The year 2012 was a significant year for DL. Many people in this area are working hard to make machines as proficient as the human brain.

In the year 2012, a DNN model named the AlexNet [1] won the ILSVRSC competition. The model was trained on a large dataset called the ImageNet.It had a very low error rate of classification. The area of deep learning gained immense popularity after this.

In the ILSVRC 2013, another efficient DNN model called ZFNet [12] won the competition. ZFNet was an enhancement over the AlexNet architecture. It was developed by tuning the hyperparameters of the AlexNet architecture.

ILSVRC 2014 was won by a CNN submitted by Google called the GoogleNet [13]. In this architecture, the Inception component was introduced. As a result of it, the model became very compact. There was also another benchmark architecture presented which clearly demonstrated that the depth of a CNN model is a crucial factor in determining its accuracy. The object localization CV task was implemented using the VGGNet DNN architecture. It became the first runner-up in the competition.

In 2015, Microsoft researcher Kaiming He developed an architecture called MSRA introducing the concept of Resnet [4] blocks and it won the ILSVRC challenge that year.

In ILSVRC 2016, ResNext [14], an enhanced ResNet [4] architecture was the first runner up in the challenge. It was developed by University of California, San Diego and Facebook AI Research.

After 2016, the deep learning annual challenge was held by Kaggle. Most of the winners who won this challenge worked in the field of CNNs or DNNs.

2.1 Neural Network

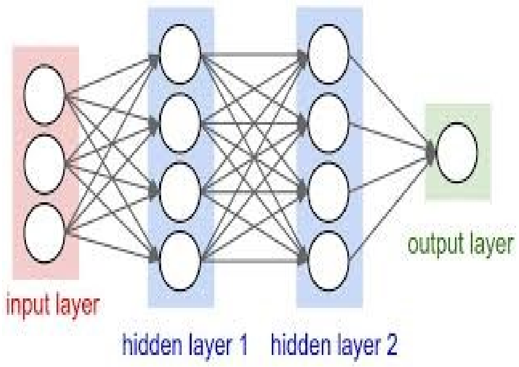


Fig. 2.1. A Neural Network [15]

The human brain has neurons which are responsible for transmitting information from one part to another. Numerous neurons are orchestrated in our brain to form a network. The nerve cells in the brain are specialized to memorize patterns or features and infer connections between objects and their characteristics. The information from one corner of the brain is transmitted to the other corner using impulses. Humans are able to learn different object properties, structures, shapes, infer various attributes and conventions through the biological neurons which interpret sensory data. Similarly, in the ML field, the ANNs and CNNs are analogous to the neurons in the human brain. They learn about image features with the help of cameras and sensors. The input image is a matrix of pixels. The NNs cluster or label the information and perform CV tasks. Fig 2.1 shows the diagram of a simple NN. The circles in the diagram are the neurons and the information is transmitted from left to right i.e from input to output through the hidden layer. A single layer in NN can be thought of as a collection of stacked nerve cells that extract some feature from the image at each level. Deep Neural Network is a neural network which has more than one hidden layer along with other visible layers connected to each other.

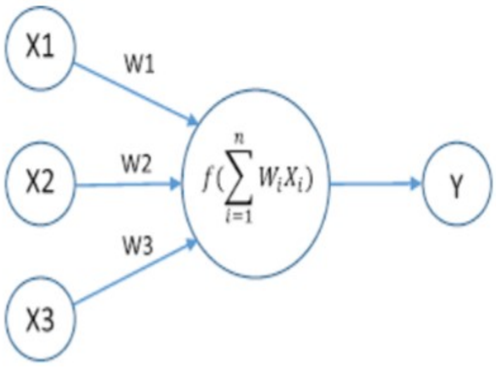


Fig. 2.2. A Simple Artificial Neuron [16]

Fig 2.2 is a diagram of a simple artificial neuron. The passing of data is through the connections made by each neurons. Each of these connections has a specific weight which is a number that determines how the neurons relate to each other. If the weight has a high positive value, it means the first neuron favours the activation of the second neuron i.e the second neuron is more likely to get activated. Here, X1,

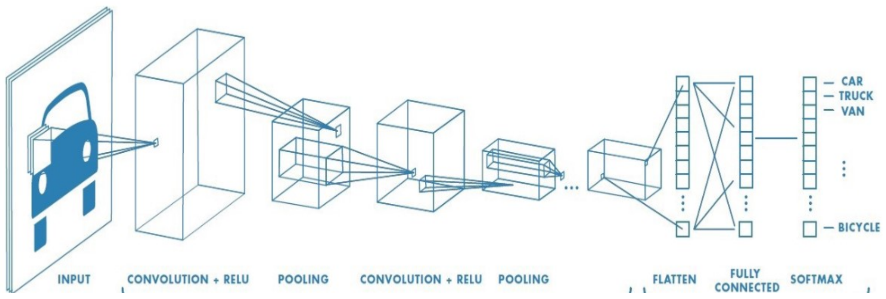


Fig. 2.3. Convolutional Neural Network [17]

X2, and X3 are the input data and these are routed through connections W1, W2, and W3 respectively, which are the connection weights. The output is obtained by first doing the scalar product of the input with suitable value of weights, and combining the result. For scaling the weights, a non-linear activation function is used. Thus, Input * Weight = Output or Prediction.

An input image or text is fed to the CNN. It passes through some convolutional layers, where features are extracted and they are combined at the end for image classification. The CNN is mainly composed of the input layer, hidden layer(s), and output layers. The hidden layers consist of pooling layers, normalization layers, and fully connected layers. These converge either by multiplication or dot product. An activation function (generally ReLU [16] [18]) conceals the input and output of these layers, succeeded by a final convolutional layer. The CNN is analogous to the cerebrum of the brain which learns to classify objects step by step by learning various characteristics for a particular class. There is a restricted area in the visual field known as the receptive field. Each nerve cell reacts to a stimulus only in this region.

2.1.1 Convolutional Layer

The input data can be an image, video or text from a dataset or real-time. The Convolutional layer is the basic building block of a CNN and is used to extract features from the input image. Initially, when the image is passed through first convolutional layers, simple features are learned. As the image passes deeper into the CNN, more complicated and intricate features are extracted. The hidden or visible layers and the kernels keep the spatial relation between pixels intact.

There are several hyperparameters that determine the performance of the CNN. These are: depth, stride and zero padding. Depth gives the output volume of the convolutional layer. It a measure of the number of filters that the network learns. These filters learn distinct attributes to provide an activation map. While performing kernel convolution, the filter slides through the image. A stride is a numerical value which determines how the filter slides through it. Zero padding is used to control the output dimension by padding zeros. The expression for calculating the output dimension is as follows:

Output Dimension= (W - F + 2P)/S + 1.

W = Input dimension.

F = Filter size.

P = Padding.

S = Stride.

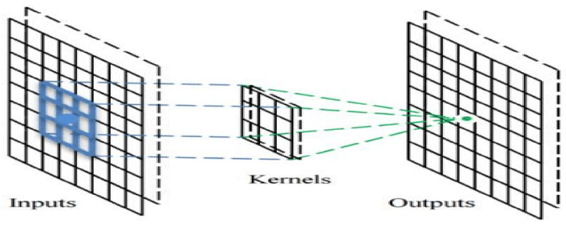


Fig. 2.4. Convolution Kernel [19]

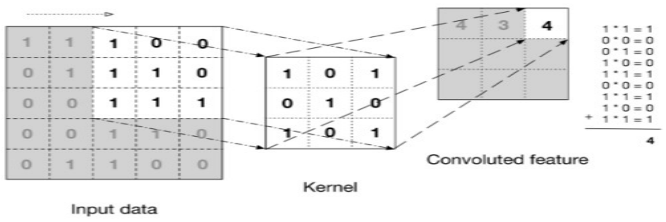


Fig. 2.5. Illustration of Convolution Operation [20]

Fig 2.4 illustrates the use of a convolutional kernel along an input channel for feature extraction. Fig 2.5 represents the convolution operation in matrix form. Here, a 5×5 matrix is an input channel is used with a 3×3 kernel or filter or feature detector for convolution operation. The filter is slid over the input image. An element wise multiplication is done between the two matrices and then they are combined (summation) to obtain a convolved feature or activation map or feature map as illustrated in Fig 2.5. Different varieties of kernels may be used but generally 1×1, 3×3, and 5×5 kernels are used. More filters means more feature extraction leading to a more efficient and accurate model in image prediction task.

2.1.2 Calculation of the Total Number of Parameters

Another critical factor to be kept in mind in when designing CNNs is the total number of parameters. The size of the model and the computation time is directly proportional to the total number of parameters, so it becomes important to calculate the number of parameters. The number of parameters depends on the dimension of the activation map and filter size. If the number of neurons in the input layer is $5\times5\times3$ and the filter size is 3 and the number of kernels is 3 then we require $5\times5\times3\times3\times3\times3 = 675$ parameters. For the sake of explanation, we have taken the image size to be 5×5 and a channel depth of 3. In reality, the image dimension is of the order 227×227 and the channel depth is 3.

2.1.3 Pooling

The other name for downsampling is Pooling. There are a different kinds of pooling layers, the common ones are Avg Pooling and Max Pooling. It produces a summarized version of the feature maps obtained after the convolution mathematical operation. Pooling has the effect of reducing the total number of parameters and overfitting.

2.1.4 Activation Function or Non-linearity

Non-linearity or activation function is a significant component of CNN. The reason for including this layer into our CNN is that it introduces some non-linearity in the network, as most of the data in real life will be non-linear. It makes a model more efficient. Some of the common activation functions are ReLU (Rectified Linear Unit [18]), ELU [21] , Sigmoid [22, 23] , tanh, LeakyReLU [24], SELU [25], and Swish. In this experiment, Drop-activation (a randomized version of ReLU) and Swish activations have been used.

2.1.5 Fully Connected Layer

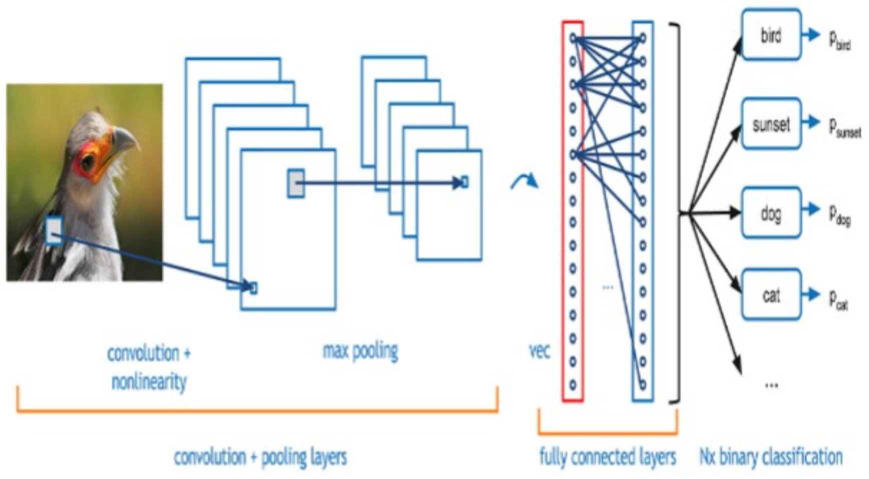


Fig. 2.6. Illustration of FC Layer Working [26]

A FC layer links each neuron from a layer to every neuron in another layer. A softmax function is used in this layer as an activation. The FC layer generally lies at the end of the network. The feature extraction has already been done by the convolutional layers earlier. Here, in the FC layer the object in the image is classified by looking at the inter-class variations within the objects. Class scores or probability values between 0 and 1 are assigned for a particular class label. In this layer, the complex features of an object are linked to a particular class. The class scores for each class are calculated based on the weights Fig. 2.6 shows the working of FC layer. There are four probable outputs such as bird, sunset, dog and cat. The category with the highest class score is the detected output.

2.2 Related DNN Architecture

2.2.1 MobileNet v1 Architecture

MobileNet v1 [11] architecture was developed by Google for embedded and mobile vision tasks. The unique feature of this streamlined DNN architecture is that it uses depthwise separable convolutions [11, 27, 28] instead of standard convolutions. The depthwise separable convolutions were first introduced into the Inception module. These convolutions are essentially an approach for making a network compact, other than compression techniques like Pruning [29], Huffman coding, etc.

The core layer of the MobileNet v1 is the depthwise separable convolution layer. These depthwise convolutions are a kind of factorized convolutions which divide a standard convolution into a depthwise and a pointwise convolution. The input channels are filtered separately in a depthwise convolution i.e a single filter is applied per input channel. The depthwise layer is followed by the 1×1 pointwise convolution layer. Here the outputs of the depthwise convolution layer are linearly combined. A standard convolution filters and integrates the outputs in one step. Unlike standard convolutions, depthwise separable convolution contains separate layers for filtering and combining the outputs respectively. This reduces the computation overhead and the model size to a great extent.

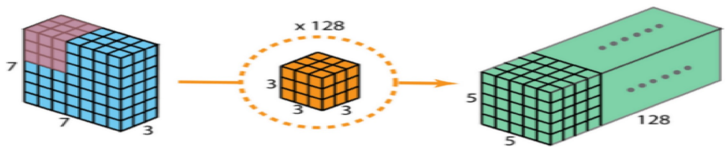


Fig. 2.7. Standard Convolutions [30]

Generally multiple kernels are applied between two NN layers. Let us assume we have 128 kernels here. Fig 2.7 shows how a standard convolution works. We convert the input layer of dimensions 7×7×3 to the output layer of dimensions 5×5×128.

The height and width are reduced, while the depth is extended. We try to achieve the same transformation using depthwise separable convolutions.

Fig. 2.8 shows a simple depthwise layer. Here, we do not use a single filter of size 3×3×3 as in the case of standard convolutions. We use the 3 filters separately, each filter having the size 3×3×1. Each filter performs convolution with one channel of the input layer. Each such convolution operation produces a map of 5×5×1 size. These maps are stacked to form a 5×5×3 image. The spatial dimensions of the output have shrunk but the depth remains constant. Fig 2.9 shows the application of the 1×1 convolution. It is applied with kernel size 1×1×3. The convolution of 5×5×3 image with each 1×1×3 filter gives a map of 5×5×1 size. Figure 2.10 shows the overall procedure of the depthwise separable convolution. It is clear now that depthwise separable convolutions need much less operations than the standard convolutions to get the same output. In case of the standard convolutions, there are 128 3×3×3 filters that are slid 5×5 times. So, the number of multiplications is 128×3×3×3×5×5 = 86,400. In case of the depthwise convolution, there are 3 3×3×1 filters that slide 5×5 times. The number of multiplications is 3×3×3×1×5×5 = 675. In case of the pointwise convolutions, there are 128 1×1×3 filters that slide 5×5 times. The number of multiplications is 128×1×1×3×5×5 = 9600. The total number of multiplications in case of the depthwise separable convolutions is 675 + 9600 = 10,275. This is about 88% less than that of the standard 2D convolutions.

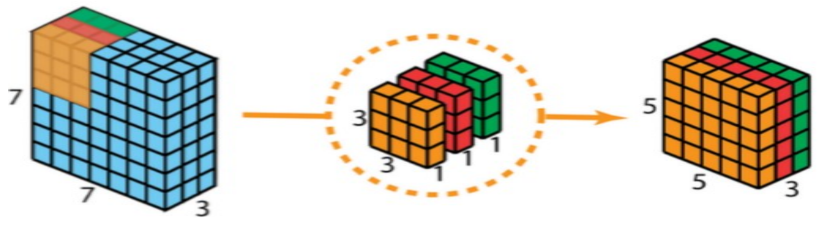


Fig. 2.8. Depthwise Layer [30]

Let us generalize the above calculations. A standard convolution takes input feature map F of dimensions $\mathbf{D_F * D_F * M}$ and outputs a feature map G of dimensions

$\mathbf{D_G*D_G*N}$. D_F is the width and the height of the input feature map and M is the input depth. D_G is the width and height of the output feature map and N is the output depth. Let D_K be the kernel width and height. The total computational cost (number of multiplications) = $\mathbf{D_F*D_F*M*N*D_K*D_K}$. MobileNet addresses the interaction

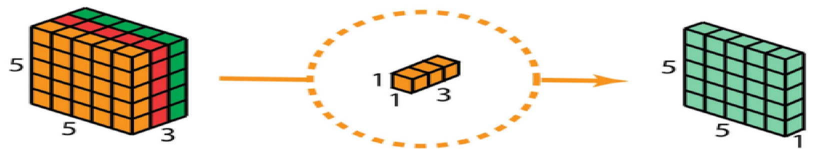


Fig. 2.9. Pointwise Layer [30]

between each and every term in the above expression. It utilizes depthwise separable convolutions to impede the interaction between the output depth and the size of the kernel. The depthwise convolution with a single filter per input channel has a computational overhead (number of multiplications): $\mathbf{D_K*D_K*M*D_F*D_F}$. The pointwise convolution which integrates all the outputs of the depthwise convolution linearly has a computational overhead (number of multiplications): $\mathbf{M*N*D_F*D_F}$. The total computational overhead of Depthwise separable convolutions = $\mathbf{D_K*D_K*M*D_F*D_F}$ + $\mathbf{M*N*D_F*D_F}$. Overall, the computation overhead decreases by one-eighth.

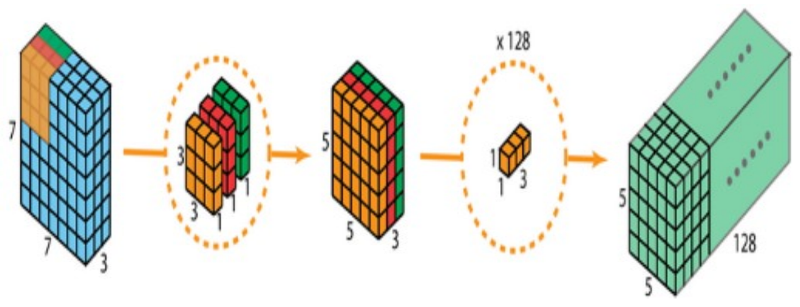


Fig. 2.10. Overall Procedure of the Depthwise Separable Convolution [30]

The MobileNet v1 architecture is mainly based on depthwise separable convolutions. It has only one fully convolution layer which is the first layer. All the 3×3 layers are succeeded by Batch-normalization and ReLU non-linear activation. Similarly, all the 1×1 layers are also followed by BN and ReLU. At last, is the FC layer which feeds into Softmax layer for the image classification task. Fig 2.11 shows a depthwise separable convolution block. Fig. 2.12 shows the MobileNet v1 architecture. The blue color block represents the first convolution layer. The numbers at the left side represents the number of filters. DWS stands for Depthwise separable convolutions. On the right end of the blocks, the stride value is mentioned. Before the FC layer, there is an Avg, pooling operation performed which decreases the value spatial resolution to 1. There are 28 layers in the baseline MobileNet v1 model, if the depthwise convolutions and the pointwise convolutions are considered separately. The 1×1 pointwise convolutions contribute about 75% of the total number of parameters, so most of the computation time is elapsed in these layers. The FC layer contains almost all the additional parameters.

Even though the baseline MobileNet v1 is a compact architecture suitable for mobile vision application, sometimes a specific application may need the model to be more compact and faster. Smaller and less expensive networks can be designed with the help of a hyperparameter known as the width multiplier. The width multiplier, denoted by α is used to trim a network uniformly at each layer. Mathematically, the parameter α is multiplied with the number of input channels M and the number of output channels N. The width multiplier [11, 28] values typically range from 1 to 0, where α=1 is the baseline setting. When α is reduced from 1 towards 0, we obtain more compact MobileNet models. The effect of using width multiplier is that the number of parameters and the computation overhead reduces quadratically. As the value of α is decreased from 1 towards 0, the model accuracy starts to fall. So, there is trade off between the model size and the model accuracy. In this experiment, the width multiplier has been used to make the improved model Ultra-thin MobileNet more compact, after increasing its accuracy above the baseline level through different methodologies. There is another hyperparameter called the resolution multiplier [2, 11, 28], denoted by ρ that also reduces the computation cost. This is applied to the input image, and not to the input and output channels as in case of α.

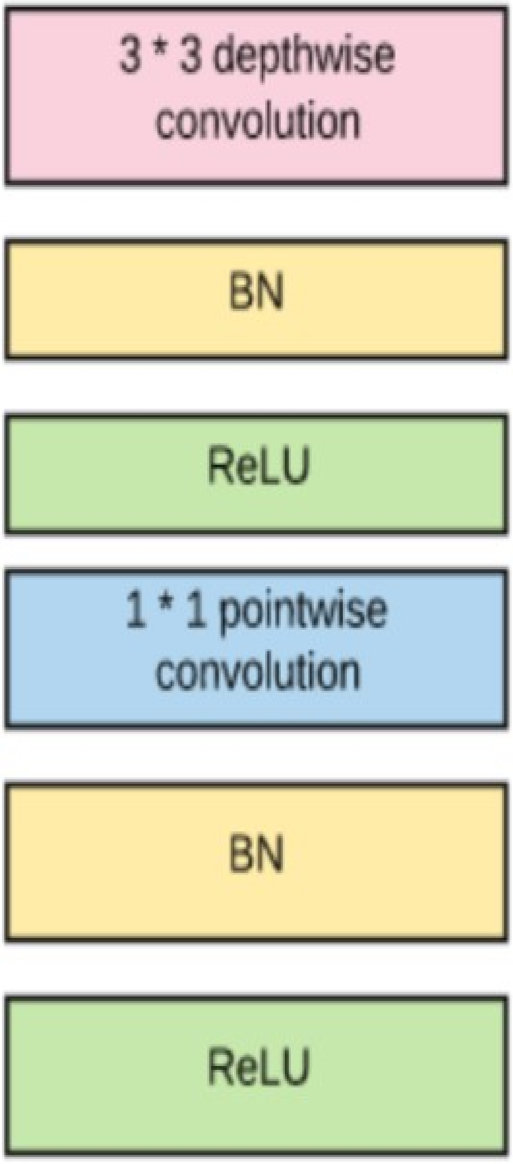


Fig. 2.11. A Depthwise Separable Convolution Block

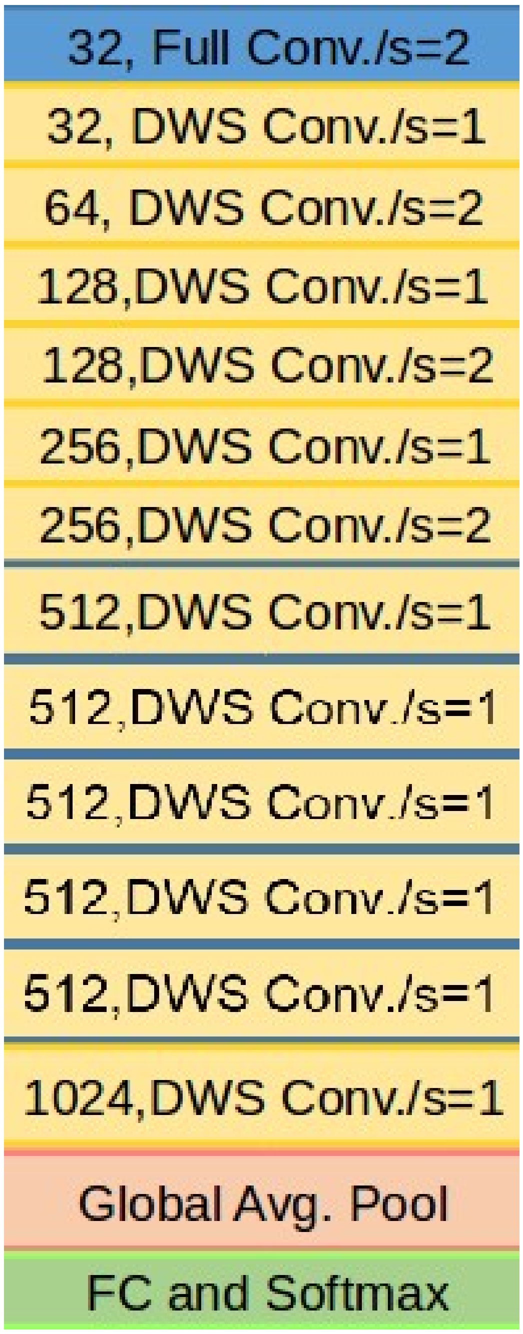


Fig. 2.12. MobileNet v1 Architecture

3. HARDWARE AND SOFTWARE REQUIRED

3.1 Software Used

(a) Python IDE- Spyder v3.6

(b) Anaconda Navigator 2.0

(c) Open-source API – Keras v2.2.0

(d) Backend framework – Tensorflow-gpu v1.11.0

(e) Livelossplot package from PyPI

3.2 Deep Learning Frameworks

3.2.1 Tensorflow

Fig. 3.1 [31] shows the logos of various DL frameworks. The TensorFlow open-source software library is used for differentiable programming and application specific data flow tasks. The name TensorFlow comes from the variety of processes that the neural networks perform on the multidimensional data arrays which are known as tensors. It provides a wide range of toolkits that allows to build different architectures at a preferred abstraction level. Tensorboard is one such tool for visualizing the network performance. Another advantage of using TensorFlow is that it provides deployment options which are production ready and the support it provides for mobile vision platforms.

Fig. 3.1. Various DL frameworks

3.2.2 Keras

Another widely used framework is the Keras. It is a high-level API written in Python, for constructing and training neural network models. It is capable of running on TensorFlow, CNTK and Theano. Some key features of this framework is it is user-friendly, modular and easily extendable. It is mainly used in classification and recognition applications. In this experiment, all the models are trained in the Keras framework.

3.2.3 PyTorch

PyTorch is an open-source machine learning library for Python based on the Torch library. This open-source library was developed by Facebook AI Research Lab. There is support for python libraries like Scikit, Numpy, Cython, etc. It is a flexible, modular and stable framework with immense support for production. PyTorch is relatively, a newer framework as compared to TensorFlow but it is slowly getting popular among researchers in the AI and Machine Learning field.

3.2.4 Caffe

Caffe was developed by Berkeley AI Research. It is written in C++ but has a Python interface. It supports different DNN architectures for image classification and segmentation application. It has support for CNN, RCNN, LSTM and neural networks which are fully connected.

3.2.5 Theano

Theano is another Python library and a compiler for optimization, mainly used to manipulate and evaluate mathematical expressions, especially those in the form of a matrix. A NumPy esque syntax is utilized to express the computations and are compiled in CPU or GPU.

3.3 Livelossplot

Livelossplot package is a python based model visualization tool which is used to visualize the loss and accuracy levels associated with training and testing a model after each epoch. It is supported by popular frameworks like Keras, PyTorch and TensorFlow. It is very significant graphical tool as it is useful in monitoring the accuracy and the losses after every epoch. The performance of the proposed architectures are demonstrated with the help of these plots. It gives information about the training and validation accuracy and the losses. It helps to distinctly compare between the performance of the baseline model and the modified models.

3.4 Hardware Used

(a) Intel i9 8th generation processor (32GB RAM)

(b) NVIDIA Geforce RTX 1080Ti GPU

(c) Memory needed for dataset and results - 4GB

(d) NXP i.MX RT1060 board

3.5 NXP i.MX RT1060 board

The NXP i.MX RT1060 is a 4-layer through-gap USB-powered PCB and at its centre lies the i.MX RT1060 hybrid MCU, highlighting NXP's propelled execution of the Arm Cortex-M7. It works at speeds up to 600 MHz to give high-performance CPU execution and great real-time response.

3.6 Specifications

(a) Processor - MIMXRT1062DVL6A.

(b) Memory - 512MB Hyper Flash, 256 MB SDRAM, 64 MB QSPI Flash.

(c) Connectivity - CAN transceivers, Micro USB OTG and Ethernet connector.

(d) Display - Camera connector, Parallel LCD connector, Microphone.

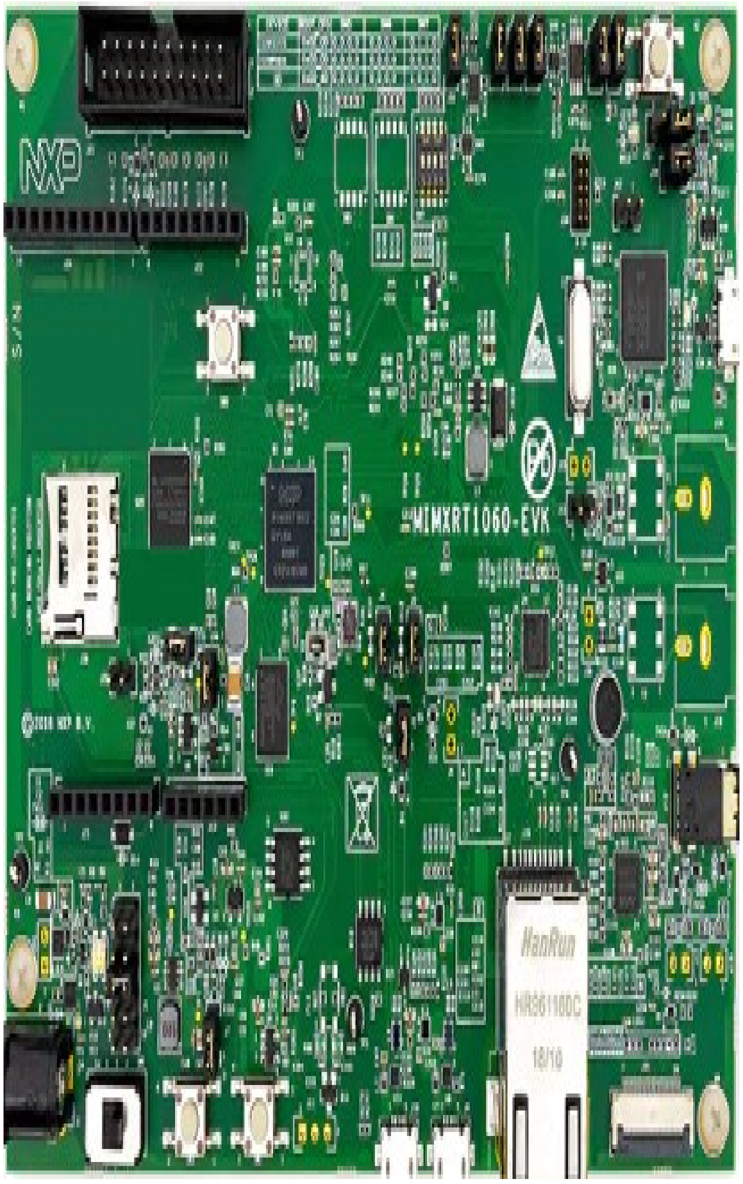


Fig. 3.2. Block Diagram of NXP i.MX RT1060 [Courtesy: NXP]

(e) Audio - Speaker connection, Audio codec, S/PDIF connector.

(e) Power - 5V DC Jack.

(f) Debug - OpenSDA having DAP link and JTAG 20-pin connector.

(g) OS support - FreeRTOS, Zephyr OS, Linux.

The MCUXpreeso is the software development package pre-configured for the i.MX RT1060 board. The toolchain is MCUXpresso IDE. For outputting the data across the MCU UART, the driver for the board's virtual COM port should be installed. The output is finally viewed in any serial terminal application like the TeraTerm.

4. DESIGN SPACE EXPLORATION TECHNIQUE

This chapter discusses the method of Design Space Exploration in detail. This is the method which has been used in developing the enhanced MobileNet architectures from the existing baseline MobileNet architecture. The following techniques improve the overall performance of DNN models:

1. Collection of more data and enhancing the quality of data improves the performance of a DNN. Data augmentation and feature selection can be used to further improve the performance.
2. Methods like weight initialization, activation functions, alteration of network topology, different optimizers, loss functions, learning rates come under the category of architecture tuning.
3. Architecture modification is another method of improving the performance where the new architecture is inspired from the literature review, and the positive features of some of the renowned DNNs.
4. Training an ensemble of many networks can be a huge boost to the performance of a model which includes combining views, stacking and combing networks.

4.1 Data Augmentation

Data augmentation is a technique of artificially increasing the training dataset size, by making modified genres of images in the dataset. It improves the training process efficiency, accuracy of the model, reduces the overfitting problem and also enhances the generalizing ability of a network. In our experiment, a data augmentation technique called Random Erasing is used to improve the overall performance. The Random Erasing technique is discussed in detail in the Chapters 6 and 7.

4.2 Weight Initialization

Weight initialization stops activation outputs from vanishing or exploding in the course of forward pass in a DNN. If the vanishing of activation outputs occur, loss gradients will either be very small, or very large to flow backwards beneficially, and the model will take a long time to converge. In this experiment, Xavier weight initialization is used. The weights are based on a Gaussian distribution. The distribution has zero mean and finite variance. With every passing layer, the variance remains fixed, which prevents the signal from exploding to a large value or vanishing to zero.

4.3 Loss Function

Loss functions measure how well a machine learning algorithm models the given data. If the estimation or prediction diverges too much from the original results, the loss function gives a large value. The Cross-entropy loss function is used, which evaluates the performance of a classification DNN model by assigning a probability value between 0 and 1.

4.4 Different Optimizers

4.4.1 SGD

SGD [32, 33] or Stochastic Gradient Descent is an iterative technique for doing optimization on an objective function with appropriate smoothness properties like differentiable and subdifferentiable properties. It performs one parameter update at a time. The SGD optimizer fluctuates a lot, as it has got a high variance due to frequent updates. As, a result the convergence to the minima becomes complicated. To solve this problem, we use Momentum combined with the SGD optimizer, which dampens oscillations. Also, Nesterov can be used to propel the SGD in the relevant path.

4.4.2 SGD with Momentum and NAG

When the gradient is continuously small, SGD can have very slow convergence. Momentum can be used to eliminate the problem and accelerate the learning. It can increase the gradient descent by taking into consideration the past gradients in the parameter update rule at each iteration for the SGD [32]. Another closely associated method to momentum is the Nesterov Accelerated Gradient [32,33] . It can be seen as a correction term for the Momentum optimization. In the Momentum optimization, the gradient is calculated with the current parameters. In case, of Nesterov, the velocity is applied to the current parameters, to find out the interim parameters. Finally, the gradient is computed with the help of the interim parameters. The parameters are updated using the similar update rule.

4.4.3 RMSProp

RMSProp [34] or Root Mean Square Propagation is an efficient optimizer as the LR is adapted for all the parameters. The main concept is to divide the LR for a weight by exponentially decaying average of magnitudes of the latest gradients for that particular weight. Sometimes, slow convergence is encountered using RMSProp.

4.4.4 Adam

Adam [35,36] is an updated version of the RMSProp optimizer. It adapts to the learning rate by using the running averages of the first moment as in the case of the RMSProp optimizer and also the second moments of the gradients are stored. In a nutshell, we can say that Adam accumulates the running average of past gradients as well as as past squared gradients. It calculates the adaptive LR of every parameter, thus eliminating manual definition of the LR. During training of DNN models, problems like slow convergence, vanishing gradient and large variance are eliminated.

4.4.5 Nadam

Nadam [32, 33], an efficient optimizer integrates Adam, RMSProp and Nesterov momentum. It accelerates the search towards the direction of minima and impedes the search in the direction of oscillations. It also does not overshoot drastically around the minima as compared to SGD and Momentum. We get the best results, in terms of accuracy, by using Nadam optimizer.

4.5 Non-Linear Activation Functions

Mainly three activation functions have been used in this research work. They are: (a) ReLU (b) Drop-activation and (c) Swish.

4.5.1 Rectified Linear Units (ReLU)

Among all the activation function, ReLU [18] is the most popular one. The mathematical expression for ReLU is max(0,z). If the input value is greater than 0, then the function is linear. If the input value is negative, the output is always a 0. ReLU can be labelled as nearly linear and as a result, it preserves many characteristics that make linear models easily optimizable. It also solves the vanishing gradient issue, and is less computationally intensive. The main demerit of ReLU is that since the ouput of the function for negative input values is 0, the ability of the DNN to fit decreases and also the negative parts are not properly mapped.

4.5.2 Drop-activation

The proposed Thin MobileNet architecture utilizes the Drop activation [37] function in place of ReLU. The non-linear function is randomly deactivated and activated during training i.e. randomness is introduced into the activation function. The non-linearity in the activation function is kept with a probability P and dropped with a probability of (1-P). Here, the non-linear activation function considered is ReLU and the way of applying ReLU to the network is modified by using Drop-Activation layer. If f(x) is the non-linear operator, x is the input and if the activation is ReLU then, f(x)=0, when x≤0 and f(x)=x with some probability, when, x≥0. This means, the function may go towards the third quadrant with some probability, even if the input is negative. This solves the above-mentioned problem with the ReLU activation function. The use of Drop-activation is discussed in more detail in Chapter 6.

4.5.3 Swish Activation

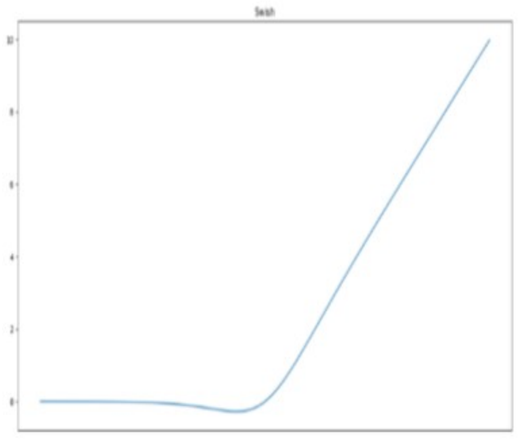


Fig. 4.1. Graph of Swish Activation Function [22]

Another activation function which consistently performs well like the ReLU is the Swish [22, 32] activation function. The Swish function is expressed as:

$$f(x) = x.\sigma(\beta.x)$$

σ = Sigmoid function.

β = A constant or trainable parameter.

Swish is a very smooth activation function. Unlike ReLU, it does not change its direction abruptly around x = 0. In Swish, the small negative input values are not zeroed out, as at times they can be relevant for seizing patterns connected to the data. This property is a clear cut advantage over the ReLU activation function.

Also, there are benefits like, the models can be optimized for convergence towards the direction of minimum loss. Fig. 4.1 shows the graphical representation of the Swish activation function. The proposed Ultra-thin MobileNet architecture uses the Swish activation function.

5. MOBILENET DNN ARCHITECTURE

5.1 Transfer Learning Technique

Usually, researchers do not train the Deep Neural Network from scratch, as training the model is a hectic task as it takes a lot of time, memory and power to train any model from scratch. There is also an issue of non-availability of datasets which are largely labelled. Generally, a pre-trained model is used and applied in a particular area. This is known as the transfer learning [38] technique.

Transfer learning is effective as many characteristics or features at low-level are common in many image classification applications. Also, a lot of memory and computation time is saved. The pre-trained model trained on a different dataset, for example, the ImageNet cannot be used in this experiment as there will be no common features in two dissimilar datasets.

5.2 Training From Scratch

The baseline MobileNet [11, 39] was trained from scratch on the CIFAR-10 dataset with Keras as the backend framework. The CIFAR-10 dataset contains 6000

Table 5.1.
MobileNet v1 Features on the CIFAR-10 dataset

Model name	MobileNet v1 baseline
Accuracy	84.30%
Model size	39.1 MB
Computation time per epoch	31s
Total number of parameters	3,239,114

images among which 5000 images are for training i.e the training set and 1000 images are for testing or validation i.e the test set. The data was trained in batches of 32 and 1563 steps-per-epoch for 200 epochs. The RMSProp optimizer is used for training the network as in the case of the original research paper [4]. The default values for the LR and momentum are used. The default momentum value is 0.9 and the default LR value is 0.001. Table 5.1 shows the accuracy, model size, computation time and the total number of parameters of the MobileNet v1 model trained on the CIFAR-10 dataset. The accuracy obtained is 84.30%. The model size is 39.1 MB, total number of parameters is approximately 3.2 million with a computation time per epoch of 31s. Fig. 5.1 shows the plot of accuracy when the MobileNet baseline model on the left side and the plot of losses on the right side. The blue line in the plot represents the training accuracy or loss and the orange line represents the test accuracy or loss. It

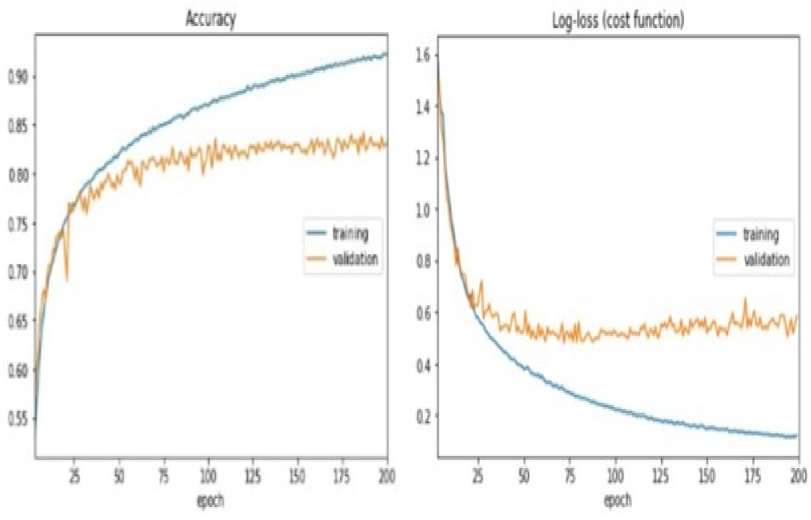


Fig. 5.1. MobileNet v1 Baseline Accuracy and Loss

is evident from the log-loss plot and the accuracy plot that the gap between the training line and the validation line is huge. The training accuracy is much higher than the test accuracy and the validation loss is much higher than the training loss. This means there is a problem of overfitting [40]. It is a problem in ML where the network predicts well with the images of the known dataset i.e the training set, but it does not predict that well when images of an unknown dataset i.e a dataset other than the training set is fed to it. The accuracy of prediction drops, thus lacking generalization ability. Also, the model is compact but still it is not deployable into microprocessors with limited space and power. The new DNN models, that is, the Thin MobileNet and the Ultra-thin MobileNet are developed using this baseline MobileNet v1 as the foundation. These enhanced models have better accuracy, model size, model speed and negligible overfitting.

6. THIN MOBILENET

The proposed architecture uses the MobileNet v1 baseline model as its foundation. The Thin MobileNet [41] has improved accuracy along-with reduced size, lesser number of layers, lower average computation time and very less overfitting as compared to the baseline MobileNet v1. The reason behind developing this model is to have a variant of the existing MobileNet model which will be easily deployable in memory-constrained MCUs. We introduce some modifications like using Separable Convolutions instead of Depthwise Separable Convolution to reduce the size of the network. Also, Drop Activation and Random Erasing methods are introduced to improve the overall performance of the model. After introducing these modifications, we make the MobileNet shallower by eliminating some layers from the network to reduce the number of parameters and computation overhead without compromising on the accuracy. The optimizer used in the baseline MobileNet is RMSProp. We replace RMSProp by the Nadam optimizer to get a better accuracy. The modified network is trained on the CIFAR-10 dataset from scratch in 32 batches and 1563 steps-per-epoch for 200 epochs. These are the following modifications:

6.1 Modification 1 - MobileNet Architecture with Separable Convolutions [42] Instead of Depthwise Separable Convolutions

A depthwise separable convolution is implemented by first performing channel-wise convolution (filtering each input channel separately) and then linearly integrating those outputs with the help of pointwise convolutions. In the baseline model, the depthwise convolution layers and the pointwise convolution layers are defined separately. In our model we use separable convolutions [42] instead of depthwise separable convolutions which combines the depthwise layer and the pointwise layer into one layer and there is no need to define them separately as two different layers.

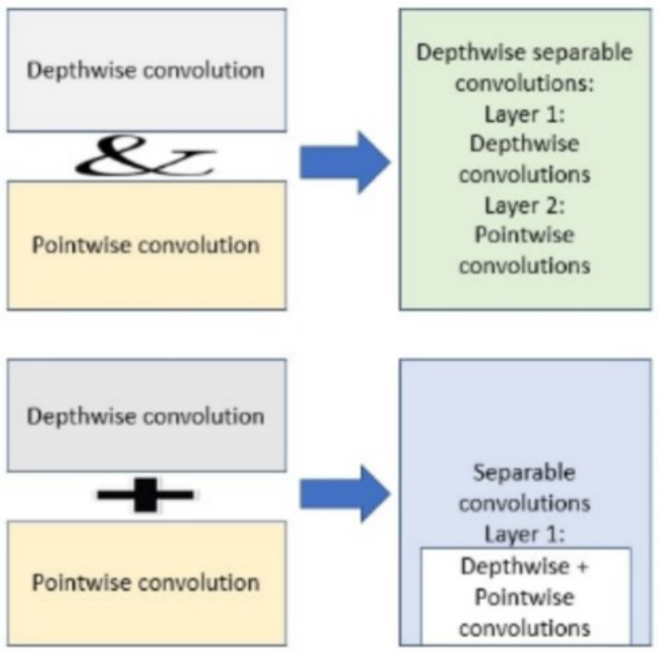

Depthwise convolution
&
Pointwise convolution
Depthwise separable convolutions: Layer 1: Depthwise convolutions Layer 2: Pointwise convolutions
Depthwise convolution
+
Pointwise convolution
Separable convolutions Layer 1:
Depthwise + Pointwise convolutions

Fig. 6.1. Depthwise Separable Convolutions and Separable Convolutions Comparison

The Keras framework is used here where the Separable Convolution 2D API is defined. Here, the pointwise initializer, pointwise regularizer and pointwise constraint for the pointwise convolution are defined inside the same init() function as the depthwise initializer, regularizer and constraints. It is a technique called deep layer aggregation. This reduces the network to 14 layers, keeping the basic functionality of the depthwise separable convolutions intact, but does not do much in increasing the accuracy of the network. The model size becomes 26.9 MB (12.2 MB less than the baseline) and the total number of parameters becomes 2,158,826. Fig. 6.1 [41] shows the difference between the core layers of the original network and the core layers of the modified network. The computation time per epoch is now reduced to 21s. Table

Table 6.1.
Network Architecture before Introducing Modification 1

Layer / Stride	Output Shape	Parameter
Input layer	32, 32, 3	0
Conv2d/s2	16, 16, 32	864
Separable conv2d/s1	16, 16, 32	1312
Separable conv2d/s2	8, 8, 64	2336
Separable conv2d/s1	8, 8, 128	8768
Separable conv2d/s2	4, 4, 128	17536
Separable conv2d/s1	4, 4, 256	33920
Separable conv2d /s2	2, 2, 256	67840
Separable conv2d /s1	2, 2, 512	133376
Separable conv2d /s1	2, 2, 512	133376
Separable conv2d /s1	2, 2, 512	266752
Separable conv2d /s1	2, 2, 512	266752
Separable conv2d /s1	2, 2, 512	266752
Separable conv2d/s2	1, 1, 512	266752
Separable conv2d /s1	1, 1, 1024	528896
Global average pool/s1	1, 1, 1024	0
FC and Softmax/s1	1,1,10	10250

6.1 [41] shows the modified deep neural network architecture with the output shape of the activation maps and the number of parameters associated with each layer.

6.2 Modification 2 - MobileNet Architecture with Drop- Activation Layers Instead of ReLU

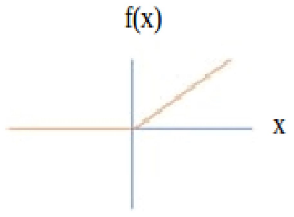


Fig. 6.2. Graph of Standard ReLU Function

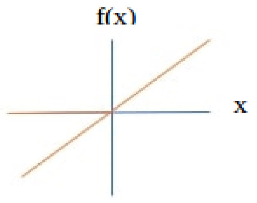


Fig. 6.3. Graph of Drop-Activation Function

Regularization [43, 44] has been an important part of Deep learning networks. Sometimes, regularizations [45] individually work quite well but when they are combined, they do not enhance the overall performance of the network. For example, if we are using Batch normalization and Dropout [46,47] in our model, the performance drops as Batch normalization requires the statistical variance should be same in both training and testing scenarios. Dropout changes the variance of the layers output when the model is in testing phase after its training phase. To make our model more robust, accurate and compatible to other regularization techniques, the non-linear activation function ReLU is replaced by Drop-Activation [37] layers. The non-linear function is randomly deactivated and activated during training i.e. randomness is introduced into the activation function. The nonlinearity in the activation function is kept with a probability P and dropped with a probability of (1-P). Here, the non-linear activation function considered is ReLU and the way of applying ReLU to the network is modified by using Drop-Activation layer.

Suppose, f(x) is the non-linear operator. If x is the input and if the activation is ReLU then, f(x)=0, when $x \leq 0$ and f(x)=x, when, $x \geq 0$. Fig. 6.2 [41] illustrates that if the input is negative, then in case of standard ReLU, the output is zero (the graph does not go to the third quadrant). Fig. 6.3 [41] illustrates that in case of drop-activation, suppose if the input is negative having the probability P=0.75, the output is zero as in normal ReLU, but the identity function I is also used with a probability of 0.25. That means, the graph may go towards the third quadrant with a probability of 0.25. Thus, we switch between standard ReLU (75%) and Identity mapping function (25%). The value of drop-activation probability P should be somewhere in between 0 and 1, since P=1 means all the non-linearities have been kept and P=0 means all the non-linearities have been dropped. In the testing phase, we average the realizations of P and get a deterministic non-linear function as a result. We use this function for testing. Mathematically speaking, we calculate the expectation of the equation of the standard non-linear function we are using during training, for example, ReLU in this case and get Leaky ReLU [48] with slope (1-P) as our deterministic non-linear function for the testing phase.

The advantages are that, Drop-activation technique increases the accuracy of the model to 85.14% and reduces the overfitting problem present in the baseline architecture. The difference between the training and testing accuracy is only 0.2 now. It is also compatible with other training methods like Batch normalization and regularization techniques like data augmentation.

6.3 Modification 3 - Use of Random Erasing in the Network

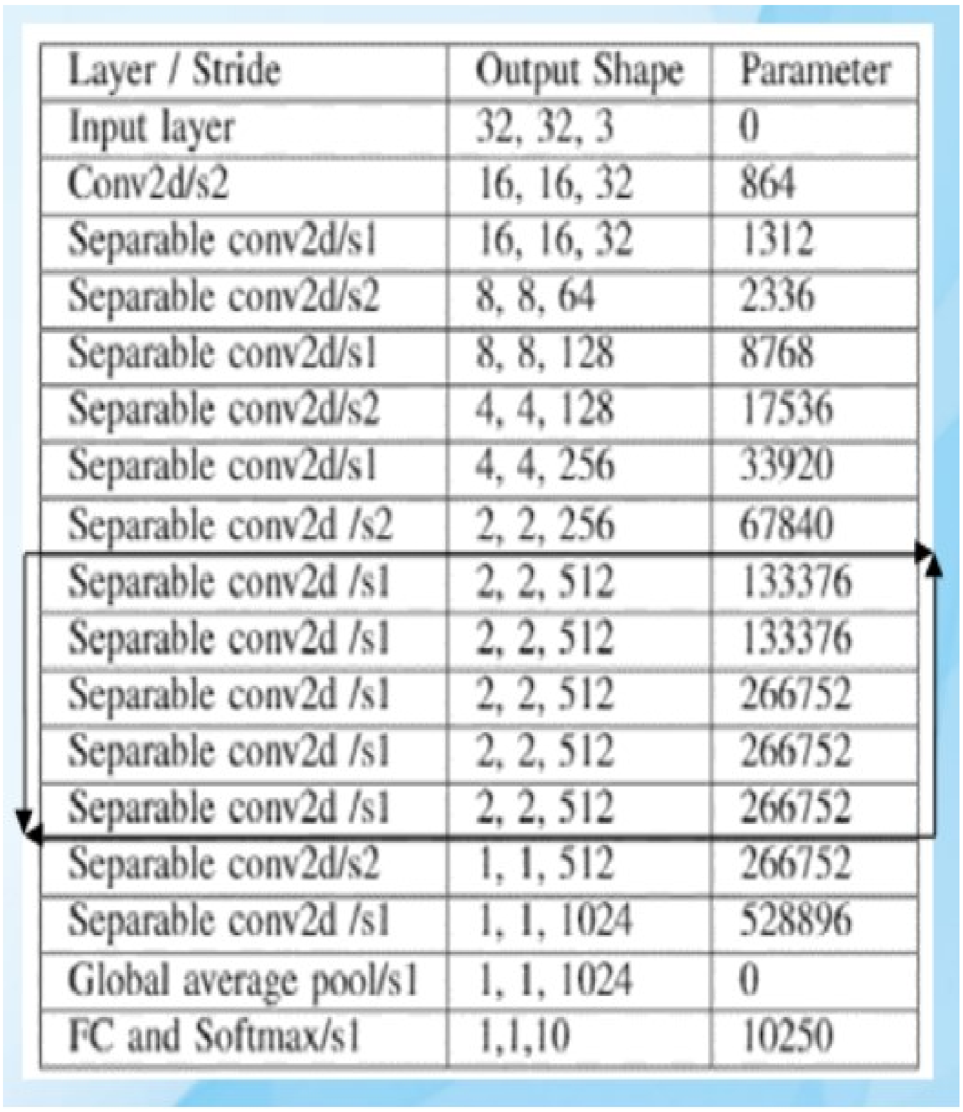
Random erasing [49] is a kind of data augmentation [50, 51] method where we select rectangular regions in an image I in a mini-batch randomly, and erase the pixels of that region and substitutes it with random values. It enhances the ability of generalization of a convolutional neural network. When some regions of an object in an image are occluded, a CNN model can be unsuccessful in recognizing the object from its global structure due to poor generalization power. To curb this problem, Random erasing [49] technique was established. Random erasing has the following input parameter values (base setting) [49] in our model:

1. Probability of erasing p = 0.5.
2. Maximum erasing area ratio S_h = 0.4.
3. Minimum erasing area ratio S_l = 0.02.
4. Erasing aspect area ratio r_e = 0.3.
5. Erasing aspect ratio range = [0.3,3.33].

The Erasing area ratio is equal to S_e/S, where S is the area of the original image and S_e is the erased area. It helps us to further enhance the accuracy to 85.21% [41] and to reduce the overfitting problem in our model. Now, the difference between the validation loss and the training loss is roughly 0.1 which is an acceptable value in any CNN object recognition model. There is a little increase in computation time per epoch which is currently 23s but still, it is much less than that of the baseline architecture which has this value as 31s.

6.4 Modification 4 - Eliminating Unnecessary Layers

We can make our architecture shallower by eliminating some layers which are repetitive [11] or redundant as they significantly increase the computation overhead by largely increasing the total number of parameters. The five layers with output shape (2,2,512) [41] that is layers, 9 to 13 as illustrated in Table 6.1 and Fig. 6.4 [41], are eliminated as they contribute about 41% of the total number of parameters. Here, the output shape (2,2,512) indicates that both the width and height of the output map is 2 and the depth is 512. This drastically reduces the model size to 9.9 MB without a decrease in the accuracy. The accuracy does not decrease because drop-activation function and the random erasing techniques which have been applied earlier compensate for the loss in accuracy.



Layer / Stride	Output Shape	Parameter
Input layer	32, 32, 3	0
Conv2d/s2	16, 16, 32	864
Separable conv2d/s1	16, 16, 32	1312
Separable conv2d/s2	8, 8, 64	2336
Separable conv2d/s1	8, 8, 128	8768
Separable conv2d/s2	4, 4, 128	17536
Separable conv2d/s1	4, 4, 256	33920
Separable conv2d /s2	2, 2, 256	67840
Separable conv2d /s1	2, 2, 512	133376
Separable conv2d /s1	2, 2, 512	133376
Separable conv2d /s1	2, 2, 512	266752
Separable conv2d /s1	2, 2, 512	266752
Separable conv2d /s1	2, 2, 512	266752
Separable conv2d/s2	1, 1, 512	266752
Separable conv2d /s1	1, 1, 1024	528896
Global average pool/s1	1, 1, 1024	0
FC and Softmax/s1	1,1,10	10250

Fig. 6.4. Eliminating Layers with Redundant Output Shape

Table 6.2.
Thin MobileNet Architecture

Layer / Stride	Output Shape	Parameter
Input layer	32, 32, 3	0
Conv2d/s2	16, 16, 32	864
Separable conv2d/s1	16, 16, 32	1312
Separable conv2d/s2	8, 8, 64	2336
Separable conv2d/s1	8, 8, 128	8768
Separable conv2d/s2	4, 4, 128	17536
Separable conv2d/s1	4, 4, 256	33920
Separable conv2d /s2	2, 2, 256	67840
Separable conv2d/s2	1, 1, 512	133376
Separable conv2d /s1	1, 1, 1024	528896
Global average pool/s1	1, 1, 1024	0
FC and Softmax/s1	1,1,10	10250

6.5 Modification 5 - Using Nadam Optimizer

Nadam [35, 36] is Nesterov Adaptive Moment Estimation. Nadam combines the positive effects of three optimizers: RMSProp [34], Adam [36] and Nesterov momentum [52, 53]. In the baseline version, RMSProp is used. In this modified, model Nadam is used instead. When RMSProp is used, the problem of slow convergence is encountered. Adam perfoms better than RMSprop but sometimes it does not converge to an optimum solution. Nadam accelerates the convergence towards the local minima. The default values for the exponential decay rate for the 1st moment estimates and the exponential decay rate for the exponentially weight infinity norm are used i.e $\beta1 = 0.9$ and $\beta2 = 0.999$. After introducing Modifications 1, 2, 3, 4 and 5 we get the Thin MobileNet DNN architecture. Table 6.2 [41] shows the Thin MobileNet architecture with its different layers, the output shape at each layer and the number of parameters contributed by them. Fig. 6.5 shows the building blocks of the Thin MobileNet architecture with the different layers, channel depths and strides.

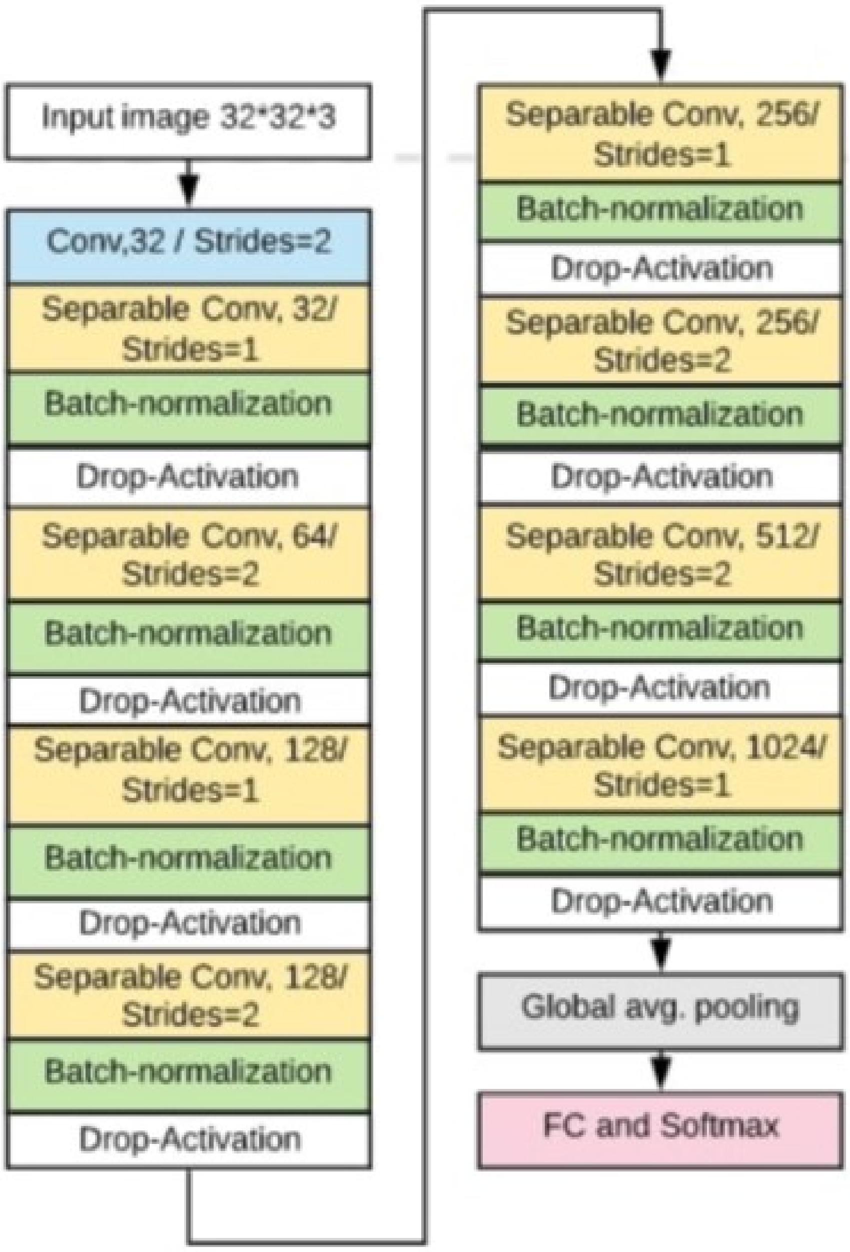


Fig. 6.5. Thin MobileNet Architecture [41]

7. KILOBYTENET AND ULTRA-THIN MOBILENET

We propose two other compact architectures inspired from the baseline MobileNet and the Thin MobileNet architecture. They have less model size and fewer number of parameters than the baseline MobileNet and the Thin MobileNet with a competitive accuracy. Design Space Exploration of the baseline MobileNet model makes it compact and less memory intensive. We propose some modifications again to develop the KilobyteNet and the Ultra-thin MobileNet architecture.

7.1 KilobyteNet Architecture Development

There are nine modifications introduced on the baseline MobileNet, four of which are already done for developing the Thin MobileNet architecture [41].

7.2 Modification 1 - Separable Convolutions Instead of Depthwise Separable Convolutions

This modification is same as the Modification 1 which we had introduced for developing the Thin MobileNet architecture. Section 6.1 of Chapter 6 discusses this modification in more detail. While keeping the basic functionality of depthwise separable convolutions the same, the number of layers is reduced to 14 which is half the number of layers in the baseline MobileNet that has 28 layers. The size of the network becomes 26.9 MB from 39.1 MB. The total number of parameters now is 2.1 M. The computation time per epoch decreases to 21s from 31s. Table 7.1 [54] shows the modified architecture. Fig. 7.1 [54] shows the difference between Depthwise separable convolutions and the Separable convolutions.

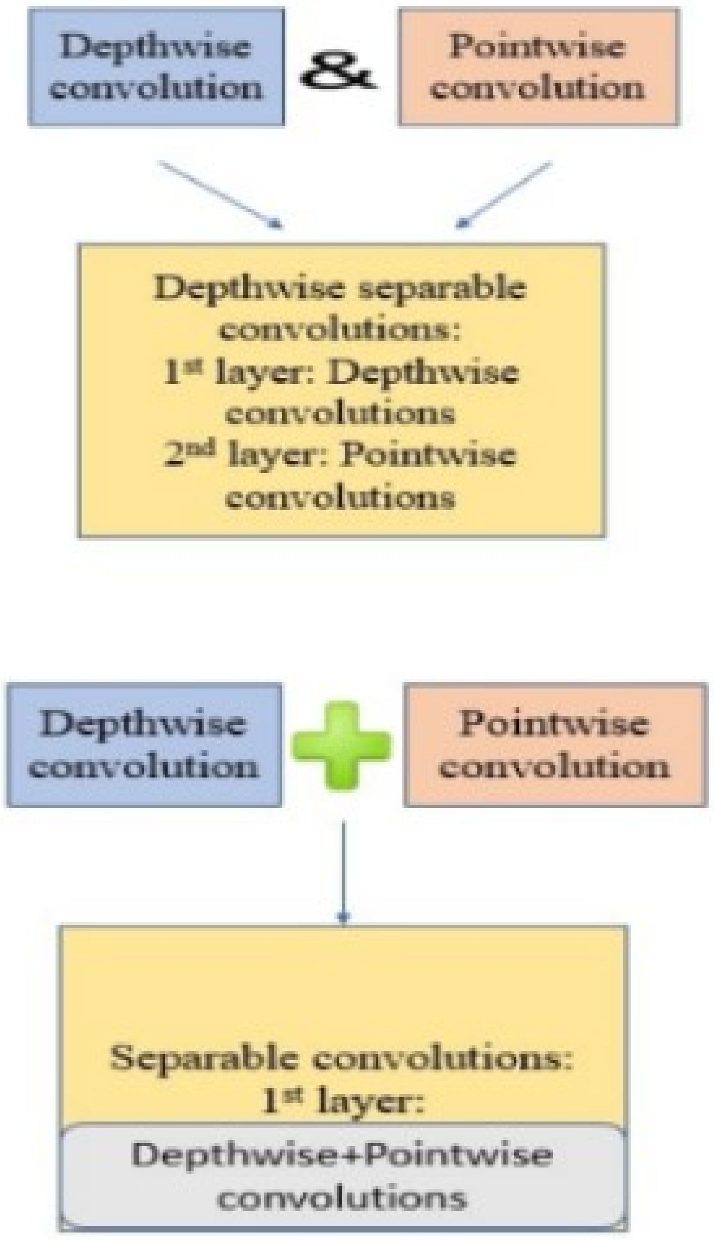

Depthwise convolution
&
Pointwise convolution
Depthwise separable convolutions:
1st layer: Depthwise convolutions
2nd layer: Pointwise convolutions
Depthwise convolution
Pointwise convolution
Separable convolutions:
1st layer:
Depthwise+Pointwise convolutions

Fig. 7.1. Depthwise Separable and Separable Convolutions

Table 7.1.
Modified Network Architecture after Introducing Modification 1

Layer / Stride	Output Shape	Parameter
Input layer	32, 32, 3	0
Conv2d/s2	16, 16, 32	864
Separable conv2d/s1	16, 16, 32	1312
Separable conv2d/s2	8, 8, 64	2336
Separable conv2d/s1	8, 8, 128	8768
Separable conv2d/s2	4, 4, 128	17536
Separable conv2d/s1	4, 4, 256	33920
Separable conv2d /s2	2, 2, 256	67840
Separable conv2d /s1	2, 2, 512	133376
Separable conv2d /s1	2, 2, 512	133376
Separable conv2d /s1	2, 2, 512	266752
Separable conv2d /s1	2, 2, 512	266752
Separable conv2d /s1	2, 2, 512	266752
Separable conv2d/s2	1, 1, 512	266752
Separable conv2d /s1	1, 1, 1024	528896
Global average pool/s1	1, 1, 1024	0
FC and Softmax/s1	1,1,10	10250

7.3 Modification 2 - Use of Random Erasing Data Augmentation Method

This modification is same as the Modification 3 which we had introduced for developing the Thin MobileNet architecture in Chapter 6 Section 6.3. This is a data augmentation method which increases the abstraction power of a model. These input parameter values are the same as that which we have used in the case of the Thin MobileNet architecture:

1. Probability of erasing p = 0.5.
2. Minimum erasing area ratio $S_l = 0.4$.
3. Maximum erasing area ratio $S_h = 0.02$.
4. Erasing aspect area ratio $r_e = 0.3$.
5. Erasing aspect ratio range = [0.3,3.33].

S_e/S = Erasing area ratio.

S = Area of the original image.

S_e = Erased area.

The accuracy of the model slightly increases [54] and the overfitting problem reduces to a large extent. The gap between the validation loss and training loss is almost 0.1 which is quite less than the baseline model. The computation time per epoch increases to 23s from 21s though [54].

7.4 Modification 3 - Removing Layers with Redundant Output Shape

This modification is same as the Modification 4 which we had introduced for developing the Thin MobileNet architecture in Chapter 6 Section 6.4. We make our model shallow by eliminating some layers (layers 9 to 13) with the same output shape of (2,2,512) [54]. This modification drastically reduces the size of the model to 9.9 MB as eliminating those five redundant layers leads to cutting off 41% of the total number of parameters.

Table 7.2.
Network Architecture after Introducing Modifications 1,2,3, and 4

Layer / Stride	Output Shape	Parameter
Input layer	32, 32, 3	0
Conv2d/s2	16, 16, 32	864
Separable conv2d/s1	16, 16, 32	1312
Separable conv2d/s2	8, 8, 64	2336
Separable conv2d/s1	8, 8, 128	8768
Separable conv2d/s2	4, 4, 128	17536
Separable conv2d/s1	4, 4, 256	33920
Separable conv2d /s2	2, 2, 256	67840
Separable conv2d/s2	1, 1, 512	133376
Separable conv2d /s1	1, 1, 728	377344
Global average pool/s1	1, 1, 728	0
FC and Softmax/s1	1,1,10	7290

7.5 Modification 4 - Altering the Channel Depth [54]

The last separable convolution block has a channel depth of 1024 which is replaced to make the network more compact. We substitute a separable convolution block with depth 728 instead of the block with channel depth 1024 to reduce the number of parameters by 0.2 million and hence, the size of the model further reduces by 1.9 MB. The total number of parameters contributed by the layer with depth 1024 is approximately 0.5 million and the total number of parameters contributed by the new layer with depth 728 is 0.3 million as is evident from, Table 7.2. Now, the model has a size of 8.0 MB. Table 7.2 [54] shows the network architecture obtained till now along-with the different layers, the output shape and the number of parameters associated with each layer.

7.6 Modification 5 - Use of Swish Activation Instead of ReLU Activation Function

ReLU is a standard non-linear activation function that consistently performs well for almost every CNN model trained on any dataset as compared to many other non-linear activation functions. Through our experiments, we find that if ReLU which was used in the baseline MobileNet, is replaced by the Swish [55] activation function, the accuracy of the model increases to 85.60%. Hence, we proceed with Swish instead of ReLU. The Swish function is defined to be:

$$x.\sigma(\beta.x)$$

Here, σ is the sigmoid function and β is a constant or trainable parameter. When, $\beta = 0$, Swish behaves as a linear function that is $f(x) = \frac{x}{2}$. When $\beta \rightarrow \infty$, Swish behaves like the ReLU function. In our model, we obtain the best performance when we use $\beta = 1$ when Swish becomes equal to the Sigmoid-weighted Linear Unit(SiL) [55]. Each separable convolution is followed by a batch normalization layer and swish activation layer as shown in Fig. 7.2 [54].

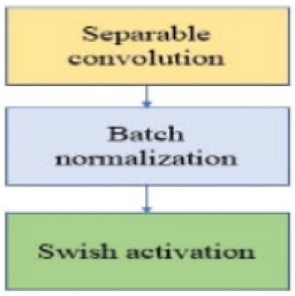


Fig. 7.2. Modified Block with Separable Convolution, Batch Normalization, and Swish Activation

7.7 Modification 6 - Dropout

The overfitting problem causes huge losses of a DNN model in terms of accuracy. They are likely to rapidly overfit a dataset due to sampling noise involved. One efficient way to deal with this problem is using ensembles of DNNs with various configurations. But this technique increases the computational expenses for training and maintenance of these models. Dropout is an alternative technique which can be used to solve the above-mentioned issues.

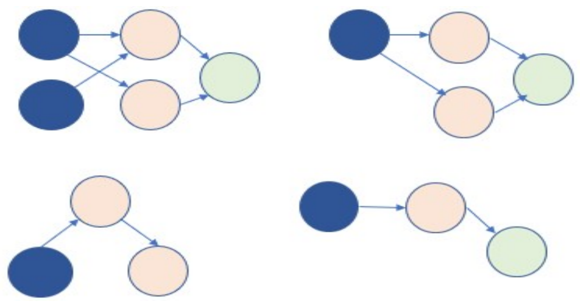


Fig. 7.3. Dropout technique

Dropout [47] is nothing but dropping out hidden and visible units in a DNN to solve overfitting issues and to maintain competitive accuracy levels. Removing a unit means eliminating a unit with all its inward and outward connections temporarily from a DNN. The method of selecting the units is random. There is a probability value p associated with it. We have chosen p=0.5 as the optimal value for our network. The dropout layer is incorporated after the last separable convolution layer with the depth 728. Every-time the network is trained with dropout, a thinner network is obtained which comprises of all the units which are retained. The usage of dropout makes the connections in a dense network sparse leading to a more compact architecture. During testing, only one DNN is used. The weights associated with this single DNN are basically scaled-down forms of the training weights. If a unit is preserved during training with a probability p, its outgoing weights are multiplied by p during testing.

This guarantees that the expected output during train time is same as the actual output during test time, thus increasing the overall accuracy of the architecture.

7.8 Modification 7 - Depth Multiplier Tuning

As discussed earlier, the MobileNet baseline uses a unique type of convolution called the depthwise separable convolution. These convolutions reduce the computation overhead by one-eight as compared to the standard convolutions. To further trim the model with a little drop in accuracy, the width multiplier hyperparameter was used. Another hyperparameter called the depth multiplier [27] can be used to produce multiple features from one input channel. It can be also said that, the value of the depth multiplier decides the number of feature maps which will be produced by each input channel. The depth multiplier is denoted by δ. The typical values of δ are 1, 2, and 4. For example, if the value of δ=2, each input channel will produce two feature maps after the convolution. If the value of δ=3, three feature maps will be produced after convolution by each input channel. The accuracy of the model increases with the increase in value of δ. But as the model accuracy increases the model size and computation overhead too. By trial and error, the value of δ=2 is chosen to be an optimal value for the KilobyteNet model. Fig. 7.4 shows the effect of using depth multiplier on the number of output channels.

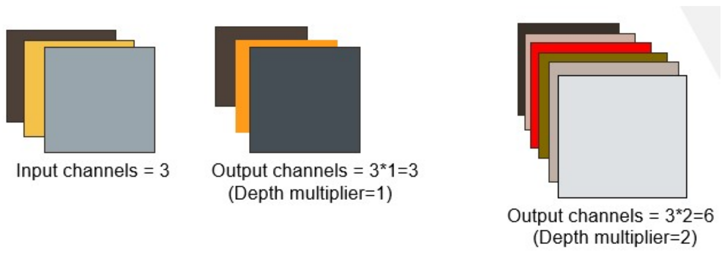


Fig. 7.4. Effect of Depth Multiplier [27]

Table 7.3.
Effect of Using Width Multiplier α

Width multiplier value	Accuracy	Model size	No. of parameters
0.75	85.95%	5.6 MB	4,85,290
0.50	82.36%	3.0 MB	2,48,269
0.25	81.59%	996 KB	74,106

7.9 Modification 8 - Tuning the width multiplier

The width multiplier α [11, 27] is used to trim a model uniformly at every layer. It reduces the model size and makes it faster. The width multiplier α should have values between 0 and 1. The baseline MobileNet has the setting $\alpha = 1$. As we decrease the value of α from 1 towards 0, the size of the model reduces as the total number of parameters decreases quadratically. The accuracy of the model starts to fall as we decrease the value of α. Table 7.3 shows the accuracy, model size and the number of parameters we obtain when we start playing with different values of α.

We observe, that as we reduce the value of α, the accuracy decreases and falls below the baseline level accuracy of 84.30%. We choose the value of α to be 0.25 so that we get a compact network in the kilobyte range and also maintaining a reasonable model accuracy (above 81%) at the same time.

7.10 Modification 9 - Using Nadam as the Optimizer

The Nadam optimizer is used as a method or algorithm to increase the learning rate of the DNN. As a result the overall rate of convergence of the DL algorithm towards the local minima gets increased. It performs better than other benchmark optimizers like SGD, NAG, Momentum, RMSProp and Adam. Its main highlight is that it prevents the curve from going towards oscillations. This is an effective optimizer which combines the positive effects of Adam, Nesterov and RMSProp and drastically reduces the loss or the cost function of the deep neural network model.

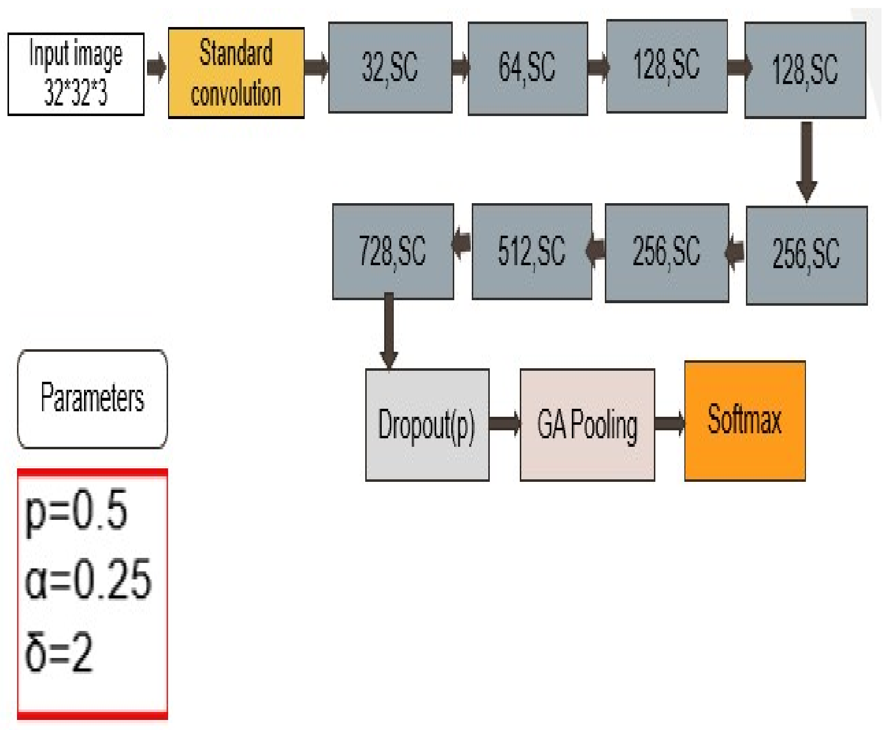


Fig. 7.5. Schematic Diagram of the KilobyteNet DNN

After introducing Modifications 1, 2, 3, 4, 5, 6, 7, 8 and 9, a new DNN architecture is obtained. It is named KilobyteNet. It is more compact as compared to the MobileNet v1 and the Thin MobileNet as it has only 10 layers in total. Fig. 7.5 shows the KilobyteNet architecture depicting its different layers. The numbers to the right of the convolution layer description represents the channel depth.

7.11 Ultra-thin MobileNet Architecture Development

Although we have obtained a compact architecture like the KilobyteNet, we still wish to further have an architecture which is small (less than 5 MB) and the accuracy almost same or higher than the baseline MobileNet. For developing the Ultra-thin MobileNet [54, 56], seven modifications have been introduced on the baseline MobileNet. These are as follows:

1. Separable convolutions instead of Depthwise separable convolutions
2. Random erasing data augmentation.
3. Eliminate redundant layers.
4. Nadam optimizer.
5. Use of Swish activation.
6. Altering the channel depth.
7. Width multiplier tuning with a different value.

Modifications 1, 2, 3, 4, 5, 6 have already been discussed for the development of the KilobyteNet architecture. The same modifications are introduced here. We eliminate the dropout technique and the depth multiplier tuning in this case to get the desired result. Also, the width multiplier is tuned to a different value for better model accuracy.

7.12 Width Multiplier Tuning

We choose the value of α between 0.75 and 0.60 such that the accuracy does not fall below the baseline level. After putting different values of α in our code, we find that when α is equal to 0.69 [54], the accuracy of the model is 84.32% which is just above the baseline level accuracy. The size of the model obtained is 3.9 MB which is less than 5 MB making it suitable for autonomous hardware deployment. Apart from reducing the number of parameters, the usage of the width multiplier makes the model faster and reduces the problem of overfitting to a large extent. Table 7.4 shows the Ultra-thin MobileNet architecture. It shows the output shape and the number of parameters associated with each layer. The depth multiplier is not used here. Fig. 7.6 shows the Schematic diagram of the Ultra-thin MobileNet.

Table 7.4.
Ultra MobileNet Architecture [54]

Layer / Stride	Output Shape	Parameter
Input layer	32, 32, 3	0
Conv2d/s2	16, 16, 22	594
Separable conv2d/s1	16, 16, 22	682
Separable conv2d/s2	8, 8, 44	1166
Separable conv2d/s1	8, 8, 88	4268
Separable conv2d/s2	4, 4, 88	8536
Separable conv2d/s1	4, 4, 176	16280
Separable conv2d/s2	2, 2, 176	32560
Separable conv2d/s2	1, 1, 353	63712
Separable conv2d/s1	1, 1, 502	180383
Global average pool/s1	1, 1, 502	0
FC and Softmax/s1	1,1, 10	5030

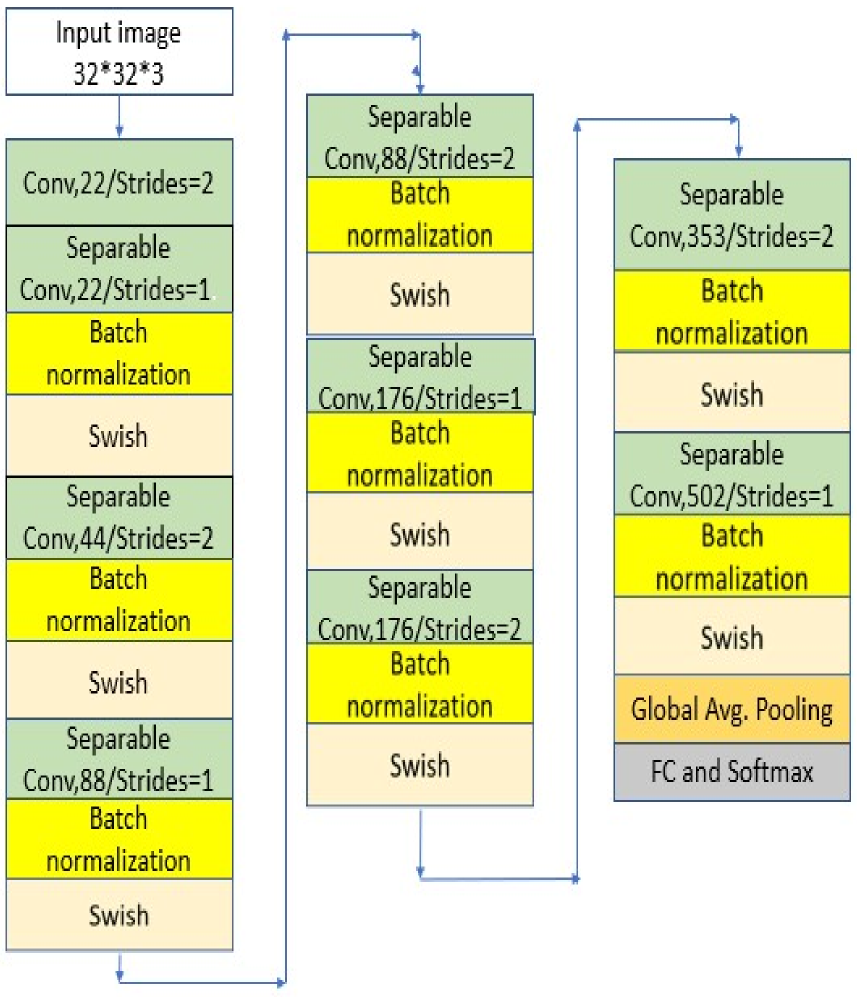


Fig. 7.6. Schematic Diagram of the Ultra-thin MobileNet DNN

8. HARDWARE DEPLOYMENT

8.1 i.MX RT1060

The Ultra-thin MobileNet model is a more balanced model in terms of model accuracy (84.32%, just above baseline) and model size (3.9 MB, less than 5 MB). It is deployed into an autonomous embedded hardware called i.MX RT1060 [56] produced by NXP semiconductors for image classification application.

8.1.1 Converting the Ultra-thin MobileNet Model into TensorFlow Lite Format

The NXP eIQ software is a machine learning software development environment which includes optimized libraries, neural network compliers and inference engines, to develop end user machine learning applications on NXP crossover processors. TensorFlow Lite is one such inference engine supported by the eIQ with a better performance and more efficient memory utilization than the TensorFlow. A model trained using the Keras backend framework is converted into TensorFlow Lite format (.tflite) using the TFLiteConverter. The python API for TFLiteConverter permits custom objects like loss functions, activation functions, etc. The integrated development environment used for the above process is Microsoft Visual Studio Code.

8.1.2 Running TFLite Model on the i.MX RT1060

The eIQ software is given as middleware in the MCUXpresso SDK for the NXP i.MX RT1060 board. It contains updated eIQ software platform plus demos. The package contains an image label demonstration for the TensorFlow Lite, which is imported into the MCUXpresso. The MCUXpresso SDK also contains UART debug console for debugging the application on the TeraTerm emulator. The model is now running in the .tflite format for image classification application. The images are captured by a camera attached to the i.MX board. The .tflite file is then converted to a C array header file that is in .h format which can be dumped into an embedded project. This .h file is utilized to load the DNN model in the code using an API call. The application is built, the code is compiled and it is now executable for the i.MX platform. Lastly, it is debugged on the TeraTerm serial emulator to get the output. Fig 8.1 and Fig 8.2 shows the flowchart for running the model on the i.MX RT1060 processor with and without camera respectively.

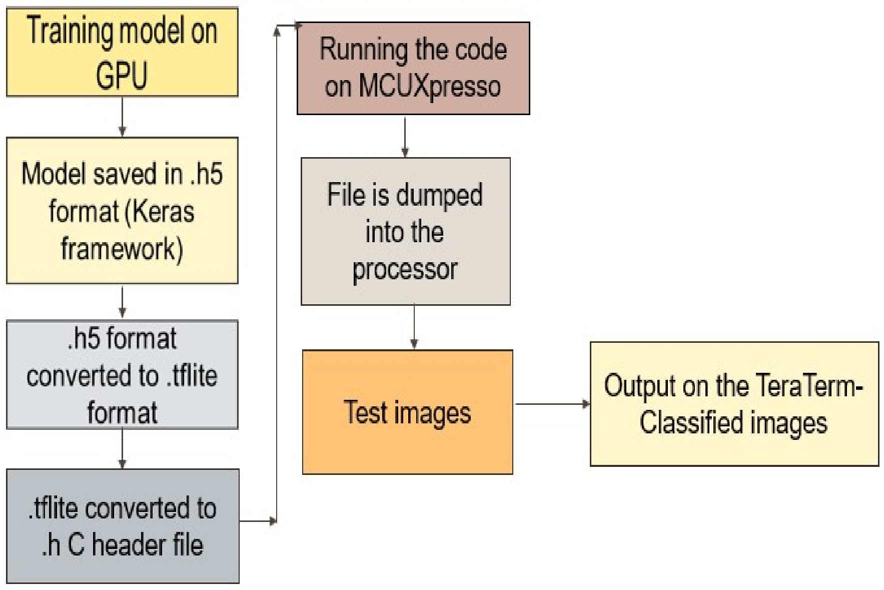

Training model on GPU
Model saved in .h5 format (Keras framework)
.h5 format converted to .tflite format
.tflite converted to .h C header file
Running the code on MCUXpresso
File is dumped into the processor
Test images
Output on the TeraTerm- Classified images

Fig. 8.1. Running Ultra-thin MobileNet on i.MX RT1060 with Test images

In case of running the model with camera, a MT9M114 camera module is needed with an LCD screen. The camera captures the image frame by frame. The image captured by the camera flashes on the LCD screen. Then image reshaping is done to obtain the image in the desired shape. Finally, the model running on the processor classifies the images frame by frame.

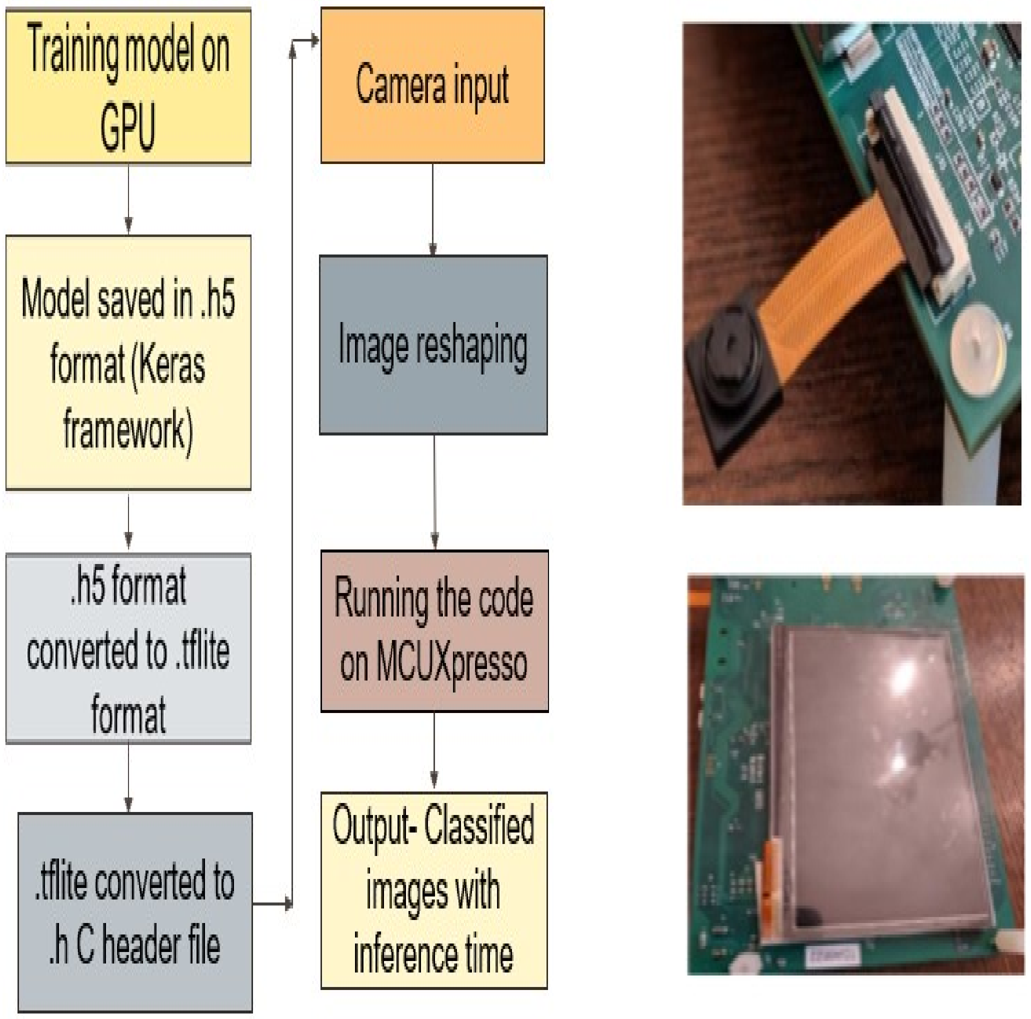

Training model on GPU
Model saved in .h5 format (Keras framework)
.h5 format converted to .tflite format
.tflite converted to .h C header file
Camera input
Image reshaping
Running the code on MCUXpresso
Output- Classified images with inference time

Fig. 8.2. Image classification with Camera [Courtesy: NXP]

9. RESULTS

In this chapter, the obtained results for the DNN architectures are discussed.

9.1 Proposed Architecture 1: Thin MobileNet

The proposed Thin MobileNet model is trained and tested from scratch on the CIFAR-10 dataset. The experimental results are based on the following parameter setting: Batch size= 32, Steps-per-epoch= 1563, Epochs= 200, Width multiplier α value= 1, Loss function= Cross-entropy. The results are visualized using the Livelossplot package which provides live loss and accuracy graphs after each epoch. Fig. 9.1 shows the plot of accuracy obtained for the Thin MobileNet architecture. Fig. 9.2 shows the loss plot for the same. The blue line in the plots represents the training part and the orange line in the plots represents the testing part. The network has been trained from scratch on the CIFAR-10 dataset. It gives better model accuracy but training the model from scratch using small datasets like CIFAR-10, CIFAR-100 takes less time and computation cost. The trained model is saved in the specified folder in the .h5 format. This .h5 file is loaded again for testing. Table 9.1 shows the model accuracy value, model size, computation time and the total number of parameters of the Thin MobileNet model.

Table 9.1.
Thin MobileNet Features

Model	Accuracy%	Model Size(MB)	Model Speed(s)	Parameters
Thin MobileNet	85.61	9.9	14	8,14,826

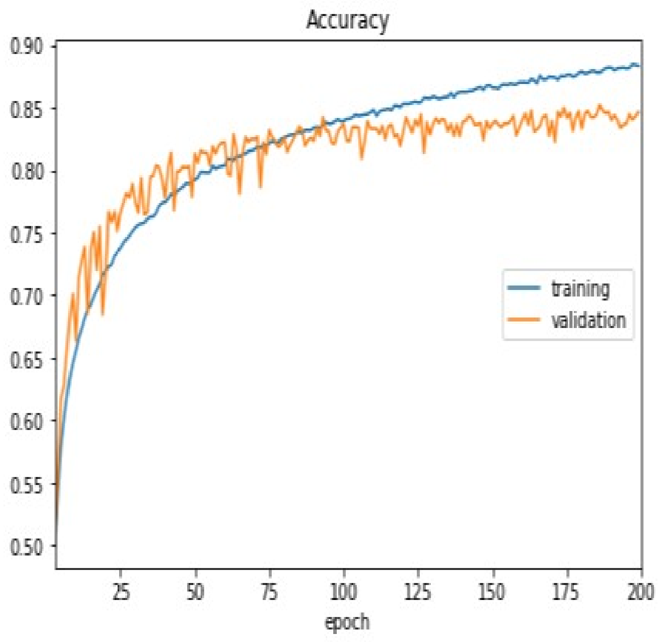


Fig. 9.1. Thin MobileNet Plot of Accuracy

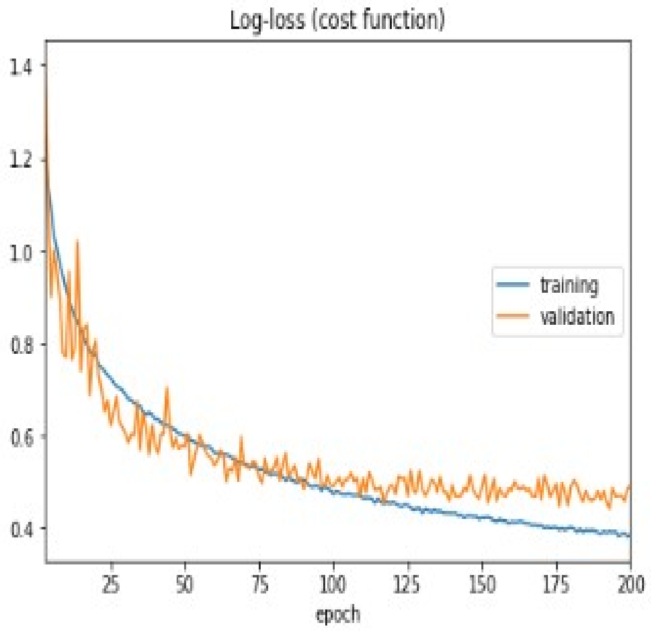


Fig. 9.2. Thin MobileNet Logloss Plot

The computation time per epoch or the model speed indicates the overhead incurred during training or testing the model after each epoch on the GPU (NVIDIA GeForce RTX 1080Ti). It is 14s for this DNN architecture. Clearly, there is a considerable improvement in the model size and model speed with a little improvement in the model accuracy too. There is almost 75% reduction in the total number of parameters, and hence the model size of the Thin MobileNet is approximately 29 MB less than the baseline architecture. The accuracy also improves by 1.31% and the overfitting problem reduces by a large extent. The proposed architecture implementation takes up to 1 hour 20 minutes. The proposed Thin MobileNet architecture is implemented using DSE techniques like separable convolutions, drop-activation function, random erasing data augmentation, redundant layer elimination, and Nadam optimizer. These modifications had been done over the baseline MobileNet v1 DNN. At last there is one global average pooling layer followed by FC layer and Softmax classifier. The width multiplier value is kept 1, as we do not want the accuracy to drop further in this case.

9.2 Proposed Architecture 2: KilobyteNet

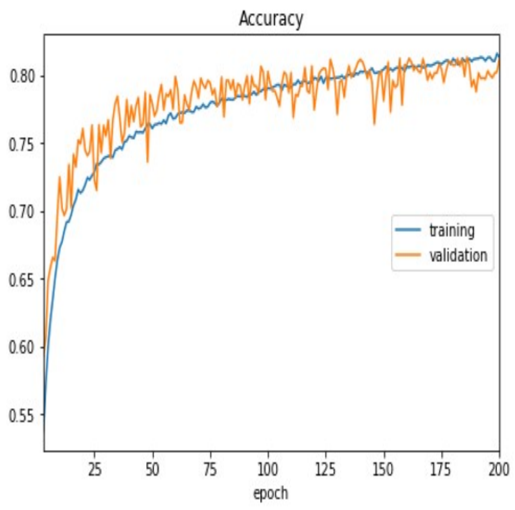


Fig. 9.3. KilobyteNet Plot of Accuracy

Table 9.2.
KilobyteNet Features

Model	Accuracy%	Model Size(MB)	Speed(s)	Parameters
KilobyteNet	81.59	0.996	14	74,106

The introduction of some more modifications on the existing baseline MobileNet v1 architecture lead to the development of a more shallower architecture called the KilobyteNet. Nine modifications are introduced into the baseline architecture to obtain this enhanced model. The proposed model is trained from scratch on the CIFAR-10 dataset.The experimental results are based on the following parameter setting: Batch size= 32, Steps-per-epoch= 1563, Weight decay=5e-4, Number of epochs= 200, Width multiplier $\alpha = 0.25$, $\delta = 2$, dropout probability p = 0.5, Loss function= Cross-entropy. The results are viewed using the Livelossplot package which provides live accuracy and loss graphs after each epoch. Table 9.2 shows the KilobyteNet features. Fig. 9.3 shows the accuracy plot for KilobyteNet. Fig. 9.4 shows the loss plot for the same.

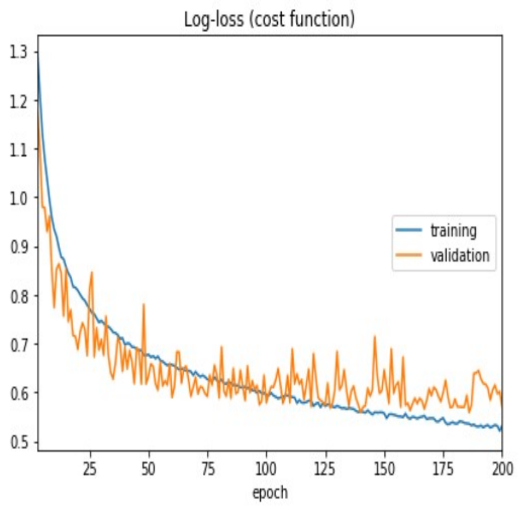


Fig. 9.4. KilobyteNet Plot of Loss

9.3 Proposed Architecture 3: Ultra-thin MobileNet

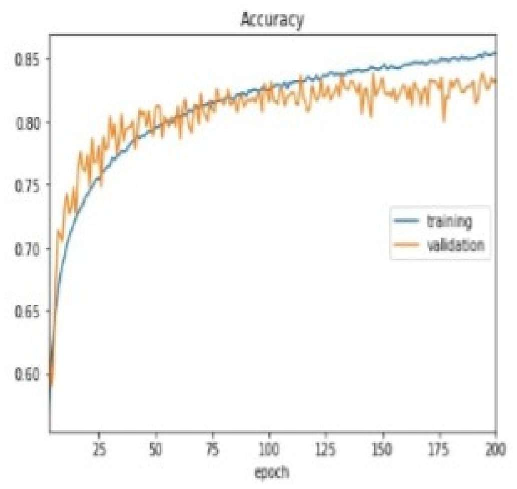


Fig. 9.5. Ultra-thin MobileNet Plot of Accuracy

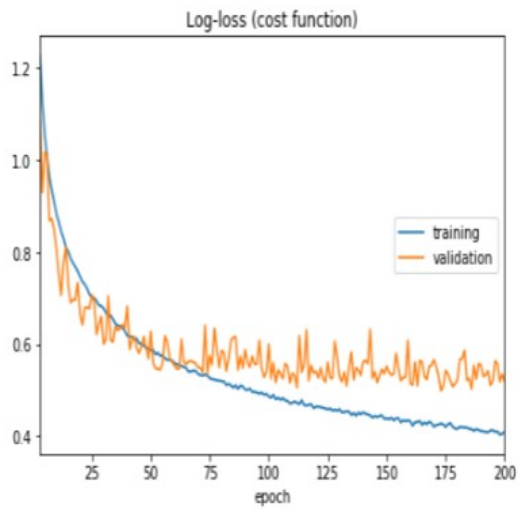


Fig. 9.6. Ultra-thin MobileNet Logloss Plot

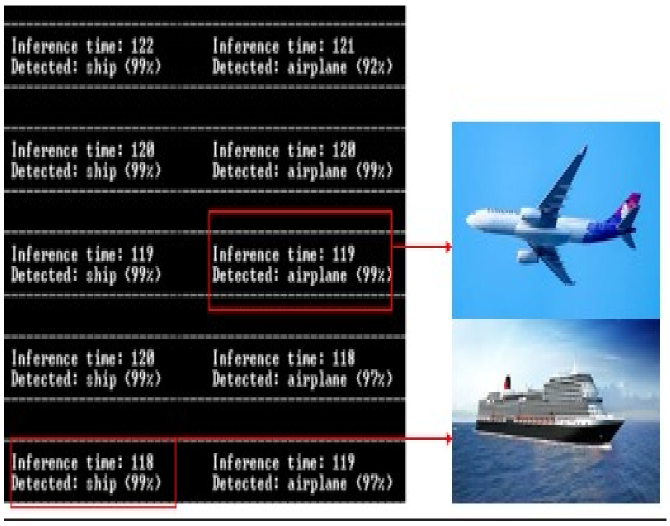
The Ultra-thin MobileNet architecture is developed by eliminating the dropout layer and the depth multiplier hyperparameter from the KilobyteNet architecture. Also, an optimum value of the width multiplier α is chosen so that a balanced model

Table 9.3.
Ultra-thin MobileNet Features

Model	Accuracy%	Model Size(MB)	Speed(s)	Parameters
Ultra-thin MobileNet	84.32	3.9	16	3,19,095

is obtained whose accuracy is almost the same as the baseline MobileNet and the size is less than 5 MB as well so that it can be deployed into resource-constrained autonomous hardware for image classification application. The proposed model is trained from scratch on the CIFAR-10 dataset.The experimental results are based on the following parameter setting: Batch size= 32, Steps-per-epoch= 1563, Weight decay=5e-4, Number of epochs= 200, Width multiplier α = 0.69, Loss function= Cross-entropy. Fig. 9.5 shows the plot of accuracy for the Ultra-thin MobileNet model. Fig. 9.6 shows the loss plot for the same. Table 9.3 shows the Ultra-thin MobileNet features.

9.4 Deployment of Ultra-thin MobileNet into i.MX RT1060

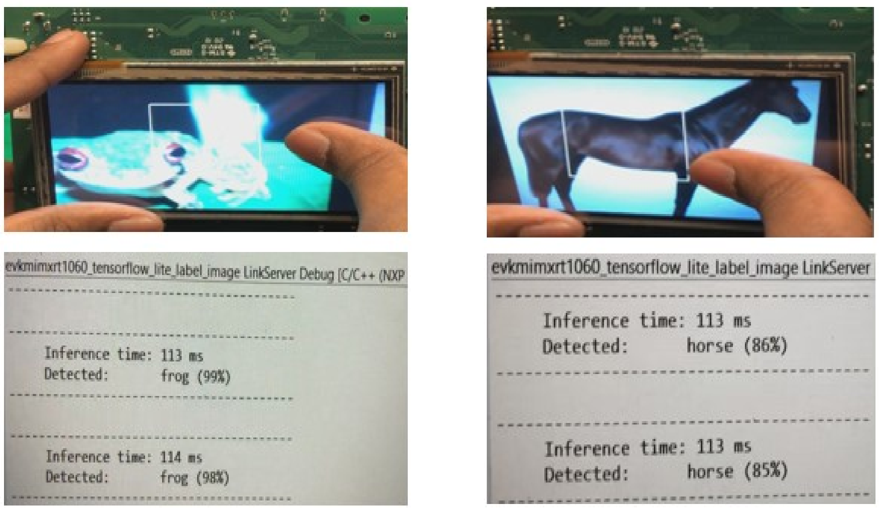


```
Inference time: 122          Inference time: 121
Detected: ship (99%)         Detected: airplane (92%)

Inference time: 120          Inference time: 120
Detected: ship (99%)         Detected: airplane (99%)

Inference time: 119          Inference time: 119
Detected: ship (99%)         Detected: airplane (99%)

Inference time: 120          Inference time: 118
Detected: ship (99%)         Detected: airplane (97%)

Inference time: 118          Inference time: 119
Detected: ship (99%)         Detected: airplane (97%)
```

Fig. 9.7. Image Classification on NXP i.MX RT1060

The Ultra-thin MobileNet DNN is deployed into the i.MX RT1060 board for image classification application by first converting the model into the .tflite format and then again converting it to the .h format as mentioned in Section 8.7. The image classification application is done for both the cases, without camera and with camera. Firstly, the model is run on the i.MX RT 1060 processor without camera input. The input are the test images and the output are the correctly detected images which are viewed on the TeraTerm window. Fig. 9.7 [56] shows the TeraTerm output.The inference time is the time taken by the DNN to classify the image. Various images of ship and airplane is fed to the network and as the results show, they are detected accurately in an average inference time of 115ms. The percentages shown inside the brackets is the confidence level with which the images are classified.

To classify images from a screen, the MT9M114 camera module is attached to the processor and focused in front of a computer screen. An LCD is attached to the processor to view the frames captured by the camera. Fig. 9.8 shows the results obtained for two classes: frog and horse. Both the classes are accurately detected at about 114ms.



```
evkmimxrt1060_tensorflow_lite_label_image LinkServer Debug [C/C++ (NXP
--------------------------------------

--------------------------------------
     Inference time: 113 ms
     Detected:       frog (99%)
--------------------------------------

--------------------------------------
     Inference time: 114 ms
     Detected:       frog (98%)
--------------------------------------
```

```
evkmimxrt1060_tensorflow_lite_label_image LinkServer
--------------------------------------
     Inference time: 113 ms
     Detected:       horse (86%)
--------------------------------------

--------------------------------------
     Inference time: 113 ms
     Detected:       horse (85%)
--------------------------------------
```

Fig. 9.8. Image Classification on NXP i.MX RT1060 with Camera

10. SUMMARY

One of the reasons why deep learning is popular today is convolutional neural networks. The motivation behind this research is designing DNN architectures suitable for real-time embedded processors. Three variants of MobileNet v1, namely Thin MobileNet, KilobyteNet and Ultra-thin MobileNet DNNs have been designed which are more efficient and flexible. First, the Thin MobileNet DNN was designed by introducing five modifications on the existing baseline MobileNet v1 architecture. It was trained and validated on the CIFAR-10 dataset. This compact architecture was then compared with the baseline MobileNet v1 architecture. It was found that the new architecture was more compact, faster and more accurate than the baseline model. Also, the overfitting problem was solved to a large extent. Then another model is designed which is the Ultra-thin MobileNet by introducing nine modifications on the baseline MobileNet, out of which four were already done in developing the Thin MobileNet. Training and testing of the above-mentioned DNN is done on the CIFAR-10 dataset. It was compared with the baseline MobileNet and the Thin MobileNet. It is even more compact than the Thin MobileNet. The accuracy is slightly lower (about 2.30%) than the baseline MobileNet. It is much faster than the baseline model, having a model speed almost comparable to that of the Thin MobileNet DNN. The overfitting problem was almost negligible in this case. Since, the accuracy of the KilobyteNet was less than the baseline MobileNet, another architecture the Ultra-thin MobileNet was developed by introducing some changes to the KilobyteNet model, to strike a good balance between the model accuracy and the model size. It is compact (less than 5 MB) and slightly accurate than the baseline MobileNet. Its model accuracy and size lies between the Thin MobileNet amd the KilobyteNet model. All the architectures were designed using a special type of convolution called the separable convolutions and other Design Space Exploration techniques. The Ultra-thin MobileNet is then deployed into an embedded processor, NXP i.MX RT1060 for image classification application. The input images (with and without camera) are classified with good confidence levels in about 115 ms which makes the DNN safe and reliable for autonomous applications. Also, the model size is only 3.9 MB which makes it easily deployable into a resource constrained processor like the NXP i.MX RT1060. Chapter 9 discusses the experimental results for the proposed DNN architectures and the deployment of Ultra-thin MobileNet into the i.MX hardware. Here is a brief summary of the research work:

1. Proposed three DNN architectures: (a) Thin MobileNet, (b) KilobyteNet and (c) Ultra-thin MobileNet.
2. The models have been trained and validated on the CIFAR-10 dataset.
3. Best accuracy model – Thin MobileNet (85.61%) - 1.31% better than the baseline MobileNet (84.30%).
4. Best model size – KilobyteNet (996 KB) - 38.1 MB better than baseline MobileNet (39.1 MB).
5. Best model speed – Thin MobileNet (14s) and KilobyteNet (14s)– 17s better than baseline MobileNet (31s).
6. Most balanced model in terms of accuracy and size – Ultra-thin MobileNet (84.32% and 3.9 MB) – 0.02% more accurate than baseline MobileNet (84.30%) and 35.2 MB less than the baseline MobileNet (39.1 MB).
7. The Ultra-thin MobileNet model is deployed into an autonomous embedded processor NXP i.MX RT1060 for image classification application. The input images (test images and camera input) are accurately classified in an average inference time of 115 ms.

11. FUTURE WORK

There are many other tools and techniques to further improve the performance of these DNN architectures:

1. Analyzing errors - Sometimes, during the testing phase, the predictions done by the DNN architectures may not be upto the mark. These bad predictions must be analyzed and changes must be made to make the model more accurate.
2. Detecting dead nodes - There can be some nodes which are not as useful as some other node in the image classification task, but may contribute a lot of parameters to the DNN. These nodes can be detected and certain methods like Pruning may be applied to cut out the connections.
3. Changing datasets - The proposed models can be trained on larger datasets like the ImageNet, STL-10 and SVHN. This can increase the model accuracy and speed.
4. Controlling exploding gradients - Techniques like gradient clipping can be used to control exploding gradients.
5. Data ensembles - Used to enhance a model. It is a procedure in which multiple models are trained and then combined together to get better performance.
6. Pruning - It is the process of making a DNN sparse by eliminating those neurons which contribute very less or nothing at all to the final result.
7. CV applications - Apart from image classification task, object localization and other CV applications can also be implemented on embedded hardware.

REFERENCES

REFERENCES

[1] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

[2] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "Squeezenet: Alexnet-level accuracy with 50x fewer parameters and¡ 0.5 mb model size," *arXiv preprint arXiv:1602.07360. (Last Day Accessed: 03/23/2020).*, 2016.

[3] A. Gholami, K. Kwon, B. Wu, Z. Tai, X. Yue, P. Jin, S. Zhao, and K. Keutzer, "Squeezenext: Hardware-aware neural network design," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, 2018, pp. 1638–1647.

[4] Z. Allen-Zhu and Y. Li, "What can resnet learn efficiently, going beyond kernels?" *arXiv preprint arXiv:1905.10337 (Last Day Accessed: 03/23/2020).*, 2019.

[5] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 1–9.

[6] M. Hossain, F. Sohel, M. F. Shiratuddin, and H. Laga, "A comprehensive survey of deep learning for image captioning," *ACM Computing Surveys (CSUR)*, vol. 51, no. 6, p. 118, 2019.

[7] X. Wang, A. Shrivastava, and A. Gupta, "A-fast-rcnn: Hard positive generation via adversary for object detection," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2017, pp. 2606–2615.

[8] T. Leung and J. Malik, "Contour continuity in region based image segmentation," in *European Conference on Computer Vision.* Springer, 1998, pp. 544–559.

[9] X. Liu, Z. Deng, and Y. Yang, "Recent progress in semantic image segmentation," *Artificial Intelligence Review*, vol. 52, no. 2, pp. 1089–1106, 2019.

[10] V. Sindhwani, T. Sainath, and S. Kumar, "Structured transforms for small-footprint deep learning," in *Advances in Neural Information Processing Systems*, 2015, pp. 3088–3096.

[11] 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 (Last Day Accessed: 03/24/2020).*, 2017.

[12] M. D. Zeiler and R. Fergus, "Visualizing and understanding convolutional networks," in *European conference on computer vision.* Springer, 2014, pp. 818–833.

[13] S. Mei, Y. Wang, and G. Wen, "Automatic fabric defect detection with a multi-scale convolutional denoising autoencoder network model," *Sensors*, vol. 18, p. 1064, 04 2018.

[14] S. Xie, R. Girshick, P. Dollár, Z. Tu, and K. He, "Aggregated residual transformations for deep neural networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 1492–1500.

[15] Pyimagesearch, "A simple neural network with python and keras," *https://www.pyimagesearch.com/2016/09/26/a-simple-neural-network-with-python-and-keras/ (Last Accessed: 03/10/2020).*, 2016.

[16] N. Joshi, "Expedition ml4sec part – 1," *https://payatu.com/expedition-ml4sec-part-1 (Last Accessed: 03/10/2020).*, 2018.

[17] S. Saha, "A comprehensive guide to convolutional neural networks — the eli5 way," *https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53 (Last Accessed: 03/11/2020).*, 2018.

[18] A. F. Agarap, "Deep learning using rectified linear units (relu)," *arXiv preprint arXiv:1803.08375 (Last Day Accessed: 03/24/2020).*, 2018.

[19] C. Thomas, "An introduction to convolutional neural networks," *https://towardsdatascience.com/an-introduction-to-convolutional-neural-networks-eb0b60b58fd7 (Last Accessed: 03/13/2020).*, 2019.

[20] D. S. Batista, "Convolutional neural networks for text classification," *http://www.davidsbatista.net/blog/2018/03/31/SentenceClassificationConvNets/ (Last Accessed: 03/12/2020).*, 2018.

[21] D.-A. Clevert, T. Unterthiner, and S. Hochreiter, "Fast and accurate deep network learning by exponential linear units (elus)," *arXiv preprint arXiv:1511.07289 (Last Day Accessed: 03/26/2020).*, 2015.

[22] C. Nwankpa, W. Ijomah, A. Gachagan, and S. Marshall, "Activation functions: Comparison of trends in practice and research for deep learning," *arXiv preprint arXiv:1811.03378 (Last Day Accessed: 03/25/2020).*, 2018.

[23] M. Tanaka, "Weighted sigmoid gate unit for an activation function of deep neural network," *arXiv preprint arXiv:1810.01829 (Last Day Accessed: 03/24/2020).*, 2018.

[24] B. Xu, N. Wang, T. Chen, and M. Li, "Empirical evaluation of rectified activations in convolutional network," *arXiv preprint arXiv:1505.00853 (Last Day Accessed: 03/24/2020)*, 2015.

[25] G. Klambauer, T. Unterthiner, A. Mayr, and S. Hochreiter, "Self-normalizing neural networks," in *Advances in neural information processing systems*, 2017, pp. 971–980.

[26] A. Deshpande, "A beginner's guide to understanding convolutional neural networks," *https://adeshpande3.github.io/A-Beginner27s-Guide-To-Understanding-Convolutional-Neural-Networks/ (Last Accessed: 03/15/2020).*, 2019.

[27] H.-Y. Chen and C.-Y. Su, "An enhanced hybrid mobilenet," in *2018 9th International Conference on Awareness Science and Technology (iCAST)*. IEEE, 2018, pp. 308–312.

[28] J. Guo, Y. Li, W. Lin, Y. Chen, and J. Li, "Network decoupling: From regular to depthwise separable convolutions," *arXiv preprint arXiv:1808.05517 (Last Day Accessed: 03/26/2020).*, 2018.

[29] H.-J. Kang, "Accelerator-aware pruning for convolutional neural networks," *IEEE Transactions on Circuits and Systems for Video Technology*, 2019.

[30] K. Bai, "A comprehensive introduction to different types of convolutions in deep learning," *https://towardsdatascience.com/a-comprehensive-introduction-to-different-types-of-convolutions-in-deep-learning-669281e58215 (Last Accessed: 03/10/2020).*, 2019.

[31] AIM, "Evaluation of major deep learning frameworks," *https://analyticsindiamag.com/evaluation-of-major-deep-learning-frameworks/ (Last Accessed: 03/11/2020).*, 2018.

[32] P. Ramachandran, B. Zoph, and Q. V. Le, "Searching for activation functions," *arXiv preprint arXiv:1710.05941 (Last Day Accessed: 03/23/2020).*, 2017.

[33] J. Ma and D. Yarats, "Quasi-hyperbolic momentum and adam for deep learning," *arXiv preprint arXiv:1810.06801 (Last Day Accessed: 03/25/2020).*, 2018.

[34] T. Tieleman and G. Hinton, "Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude," *COURSERA: Neural networks for machine learning*, vol. 4, no. 2, pp. 26–31, 2012.

[35] S. Ruder, "An overview of gradient descent optimization algorithms," *arXiv preprint arXiv:1609.04747 (Last Day Accessed: 03/25/2020).*, 2016.

[36] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980 (Last Day Accessed: 03/24/2020).*, 2014.

[37] S. Liang, Y. Kwoo, and H. Yang, "Drop-activation: Implicit parameter reduction and harmonic regularization," *arXiv preprint arXiv:1811.05850 (Last Day Accessed: 03/26/2020).*, 2018.

[38] C. Tan, F. Sun, T. Kong, W. Zhang, C. Yang, and C. Liu, "A survey on deep transfer learning," in *International conference on artificial neural networks*. Springer, 2018, pp. 270–279.

[39] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "Mobilenetv2: Inverted residuals and linear bottlenecks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 4510–4520.

[40] P. L. Bartlett, "For valid generalization the size of the weights is more important than the size of the network," in *Advances in neural information processing systems*, 1997, pp. 134–140.

[41] D. Sinha and M. El-Sharkawy, "Thin mobilenet: An enhanced mobilenet architecture," in *2019 IEEE 10th Annual Ubiquitous Computing, Electronics Mobile Communication Conference (UEMCON)*, Oct 2019, pp. 0280–0285.

[42] F. Chollet, “Xception: Deep learning with depthwise separable convolutions,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 1251–1258.

[43] L. Wan, M. Zeiler, S. Zhang, Y. Le Cun, and R. Fergus, “Regularization of neural networks using dropconnect,” in *International conference on machine learning*, 2013, pp. 1058–1066.

[44] J. Kukačka, V. Golkov, and D. Cremers, “Regularization for deep learning: A taxonomy,” *arXiv preprint arXiv:1710.10686 (Last Day Accessed: 03/25/2020).*, 2017.

[45] P. Lemberger, “On generalization and regularization in deep learning,” *arXiv preprint arXiv:1704.01312 (Last Day Accessed: 03/23/2020).*, 2017.

[46] J. Ba and B. Frey, “Adaptive dropout for training deep neural networks,” in *Advances in neural information processing systems*, 2013, pp. 3084–3092.

[47] H. Wu and X. Gu, “Towards dropout training for convolutional neural networks,” *Neural Networks*, vol. 71, pp. 1–10, 2015.

[48] X. Zhang, Y. Zou, and W. Shi, “Dilated convolution neural network with leakyrelu for environmental sound classification,” in *2017 22nd International Conference on Digital Signal Processing (DSP).* IEEE, 2017, pp. 1–5.

[49] Z. Zhong, L. Zheng, G. Kang, S. Li, and Y. Yang, “Random erasing data augmentation,” *arXiv preprint arXiv:1708.04896 (Last Day Accessed: 03/24/2020).*, 2017.

[50] B. Zoph, E. D. Cubuk, G. Ghiasi, T.-Y. Lin, J. Shlens, and Q. V. Le, “Learning data augmentation strategies for object detection,” *arXiv preprint arXiv:1906.11172 (Last Day Accessed: 03/25/2020).*, 2019.

[51] Q. Xie, Z. Dai, E. Hovy, M.-T. Luong, and Q. V. Le, “Unsupervised data augmentation,” *arXiv preprint arXiv:1904.12848 (Last Day Accessed: 03/25/2020).*, 2019.

[52] I. Sutskever, J. Martens, G. Dahl, and G. Hinton, “On the importance of initialization and momentum in deep learning,” in *International conference on machine learning*, 2013, pp. 1139–1147.

[53] A. Botev, G. Lever, and D. Barber, “Nesterov’s accelerated gradient and momentum as approximations to regularised update descent,” *arXiv preprint arXiv:1607.01981 (Last Day Accessed: 03/23/2020).*, 2016.

[54] D. Sinha and M. El-Sharkawy, “Ultra-thin mobilenet,” in *2020 10th Annual Computing and Communication Workshop and Conference (CCWC)*, January 2020, pp. 0234–0240.

[55] S. Elfwing, E. Uchibe, and K. Doya, “Sigmoid-weighted linear units for neural network function approximation in reinforcement learning,” *Neural Networks*, vol. 107, pp. 3–11, 2018.

[56] S. R. Desai, D. Sinha, and M. El-Sharkawy, “Image classification on nxp i.mx rt1060 using ultra-thin mobilenet dnn,” in *2020 10th Annual Computing and Communication Workshop and Conference (CCWC)*, January 2020, pp. 0474–0480.