DESIGN SPACE EXPLORATION OF DNNS FOR AUTONOMOUS SYSTEMS

A Thesis

Submitted to the Faculty

of

Purdue University

by

Jayan Kant Duggal

In Partial Fulfillment of the

Requirements for the Degree

of

Master of Science in Electrical and Computer Engineering

August 2019

Purdue University

Indianapolis, Indiana

THE PURDUE UNIVERSITY GRADUATE SCHOOL
STATEMENT OF THESIS APPROVAL

Dr. Mohamed El-Sharkawy, Chair

Department of Electrical and Computer Engineering

Dr. Brian King

Department of Electrical and Computer Engineering

Dr. Maher Rizkalla

Department of Electrical and Computer Engineering

Approved by:

Dr. Brian King

Head of the Graduate Program

This is dedicated to my beloved mother, late Meena Duggal, father, Hari Kant Duggal, younger brother, Hridey Anand Duggal, to all of my wonderful friends, and IoT lab Collaboratory colleagues.

ACKNOWLEDGMENTS

I would specifically, like to thank my adviser Dr. Mohamed El-Sharkawy for supervising and providing resources for the thesis research. I would also, like to extend my heartfelt gratitude to the other thesis committee members Dr. Brian King & Dr. Maher Rizkalla.

I would also, like to thank the Electrical and Computer Engineering Department, especially Sherrie Tucker, for swift proceedings and assistance throughout my time at IUPUI. Additionally, I too, thank all the people at IoT Collaboratory lab for the positive working atmosphere. Lastly, I thank my family and friends for their love and support.

TABLE OF CONTENTS

Page

Page

Page

LIST OF TABLES

LIST OF FIGURES

Figure Page

ABBREVIATIONS

Adagrad	Adaptive subgradient methods for Online Learning and Stochastic Optimization
Adam	Adaptive Moment Estimation
ADAS	Advanced Driver Assistance Systems
AI	Artificial Intelligence
ANNs	Artificial Neural Networks
ASGD	Averaged Stochastic Gradient Descent
Backprop	Back propagation
BLBX2	BlueBox2.0
BSP	Board Support Packages
CMYK	Cyan, Magenta, Yellow, Black
CNN(s)	Convolutional Neural Network(s)
ConvNet(s)	Convolutional Network(s)
CV	Computer Vision
DCCN	Densely Connected Convolutional Network
DCNN(s)	Deep Convolution Neural Network(s)
DL	Deep Learning
DNN(s)	Deep Neural Network(s)
DSE	Design space exploration
E.g.	Example
ELU	Exponential Linear Unit
FC	Fully Connected
FCNN(s)	Fully Convolutional Neural Network(s)
FPGA(s)	Field Programmable Gate Array

GD	Gradient Descent
GPU(s)	Graphic Processing Unit
HPSqNxt	High Performance SqueezeNext
HSV	Hue, Saturation, Value
ILSVRC	ImageNet Large Scale Visual Recognition Competition
ISP	Image Signal Processor
LR	Learning Rate
LSTM	Long Short-Term Memory
KSIZE	Kernel Size
M	Million
Max	Maximum
Min	Minimum
NN(s)	Neural Network(s)
ReLU	Rectified Linear Unit
RGB	Red, Green, Blue
RMSprop	RMSprop
RNN(s)	Recurrent Neural Network
ROI	Region Of Interest
Rprop	Resilient back propagation
RTMaps	Real-Time Multisensor Applications
SDK	Software Development Kit
SIMD	Single Instruction Multiple Data
SGD	Stochastic Gradient Descent
SSD	Single Shot Detector
SSqnxt	Shallow SqueezeNext
SVM	Support Vector Machines
TCP/IP	Transmission Control and Internet Protocol
TF	Tensorflow
TPU(s)	Tensor Processing Unit

UAVs	Unmanned Aerial Vehicles
VGG	Visual Geometry Group
YOLO	You Look Only Once
ZFNet	Zeiler& Fergus Net

ABSTRACT

Duggal, Jayan Kant. M.S.E.C.E., Purdue University, August 2019. Design Space Exploration of DNNs for Autonomous Systems. Major Professor: Mohamed El-Sharkawy.

Developing intelligent agents that can perceive and understand the rich visual world around us has been a long-standing goal in the field of AI. Recently, a significant progress has been made by the CNNs/DNNs to the incredible advances & in a wide range of applications such as ADAS, intelligent cameras surveillance, autonomous systems, drones, & robots. Design space exploration (DSE) of NNs and other techniques have made CNN/DNN memory & computationally efficient. But the major design hurdles for deployment are limited resources such as computation, memory, energy efficiency, and power budget. DSE of small DNN architectures for ADAS emerged with better and efficient architectures such as baseline SqueezeNet and SqueezeNext. These architectures are exclusively known for their small model size, good model speed & model accuracy.

In this thesis study, two new DNN architectures are proposed. Before diving into the proposed architectures, DSE of DNNs explores the methods to improve DNNs/CNNs. Further, understanding the different hyperparameters tuning & experimenting with various optimizers and newly introduced methodologies. First, High Performance SqueezeNext architecture ameliorate the performance of existing DNN architectures. The intuition behind this proposed architecture is to supplant convolution layers with a more sophisticated block module & to develop a compact and efficient architecture with a competitive accuracy. Second, Shallow SqueezeNext architecture is proposed which achieves better model size results in comparison to baseline SqueezeNet and SqueezeNext is presented. It illustrates the architecture is

compact, efficient and flexible in terms of model size and accuracy. The state-of-the-art SqueezeNext baseline and SqueezeNext baseline are used as the foundation to recreate and propose the both DNN architectures in this study. Due to very small model size with competitive model accuracy and decent model testing speed it is expected to perform well on the ADAS systems. The proposed architectures are trained and tested from scratch on CIFAR-10 [30] & CIFAR-100 [34] datasets. All the training and testing results are visualized with live loss and accuracy graphs by using livelossplot. In the last, both of the proposed DNN architectures are deployed on BlueBox2.0 by NXP.

1. INTRODUCTION

DNN has completely revolutionized the image classification domain & dominant technology in tasks such as image recognition, object recognition and detection. DNN surpasses the traditional algorithms and human performance in terms of accuracy and detection time. In DNNs, instead of manually engineering the features, supervised learning helps to learn these features through learning algorithms and optimization techniques. In many real world applications such as ADAS, robotics, self-driving cars, and augmented reality, the recognition tasks are needed to be carried out in a timely fashion on a computationally limited platform. The general trend has been to make the DNN architectures deeper in order to achieve a higher accuracy [1, 2]. The performance of CNN has been progressing at a dramatic pace, majorly due to new ideas, algorithms, improved network architectures, powerful hardware, and larger datasets. There are several advantages of small DNN architectures such as less model update overhead, low model size, low time complexity, and better feasibility for hardware deployment. Recently, DNN achieved an astonishing benchmark accuracy of 99% with GPipe: efficient training of giant neural networks using pipeline parallelism [35]. CNN emerged out of the existing algorithms such as SIFT, HOG, etc. as a good algorithm for image classification, object recognition, object segmentation [3] and detection. CNNs have been used to learn image representations while RNNs have demonstrated the ability to generate text from visual stimuli. DNN is, however, a memory and computationally intensive algorithm. Though the DNNs are a great tool to attack image classification problems, it is computationally and memory intensive.

1.1 Context

DNN emerged out of the existing algorithms such as SIFT, HOG, etc, as a good algorithm for image classification, object recognition, and detection. DSE of CNNs, new techniques and methodologies have made DNN memory and computationally efficient. It made it suitable to deploy CNN on real-time embedded systems or autonomous systems [37, 38]. An efficient DNN ir required to ameliorate the performance of existing DNN architectures. The focus of the thesis is to propose flexible DNN architectures which can perform better with the minimum tradeoff between model accuracy, model size, and model speed. Further, the proposed architecture is deployed on a real-time embedded platform called BLBX2 by NXP using RTMaps software framework [39]. The increasing complexity of NN has also led to the development of better software and hardware architectures [8], which might further uses parallel-distributed architectures and multiple computation units for e.g. BLBX2.0, S32V234 by NXP, NVIDIAs GPUs and Intel CPUs.

DNNs or DCNNs have become the popular subsets of ML domain to improve the performance of CV, DL or ML applications [40]. However, to counter the computation and memory issues with the DNN, the technique of squeezing, expanding the network. The compact model with a competitive accuracy can be developed to cope with the limited memory and power constraints and can be used for deployment on autonomous or embedded systems. Picture a venn diagram DL is the subset of ML and further, ML is the subset of AI, now here DNN falls in the deep learning subset.

The history of **Cybernetics**, and scientists working in cognitive science or human brain neurons, this all started in the era of early 1950's, with which the DL came in actual existence around the 1940-1960s.Then, we came across **Connectionism** in the 1960s-1980s due to the the invention of algorithm called back propagation. This algorithm is currently being implemented to optimize DNNs. A influential breakthrough came when CNNs designed are able to recognize simple visual patterns [11], such as handwritten characters during the 1960s. But

after this NN community went silent for few decades as traditional algorithms again gained popularity among the scientists. Finally, again in 2006, the modern era of DL started with the creation of more complex and better NN architectures [1-3, 5-9]. In 2011, with a breakthrough in NLP, and image classification at ILSVRC competition in 2012, DL community got revived and rejuvenated. DL has won challenges beyond the reach of conventional and traditional algorithm applications area.

DL also has made a tremendous progress in CV reaching on many applications such as objects detection, localization, and tracking, pose estimation, object segmentation [3, 10], NLP, or image captioning. This was made possible by the advent in hardware and software resources like more computational and memory resources, software frameworks such as Pytorch, TF, Theano, Keras and many modern GPUs new hardware and software implementations such as Cudnn, and the large open source community involved to make better software frameworks and CNN techniques better. This encouraged a much larger community to acquire the expertise, train and test CNN rapidly.

Thus, more deeper architectures are trained on large datasets such as ImageNet to achieve better model performance. Also, already with the help of transfer learning, the trained models have shown promising results during the CNN model testing. Hence, due to transfer learning a lot of pre-trained models are readily available. Abundant of the DNN/ CNN research is made with hit and trial or a empirical oriented methodology.

Deeper architectures demand more computational resources than the shallow neural network. Also, the available small or shallow DNN architectures focus only on model size but do not consider speed. Most of DNN research, has abundantly focused on designing and developing more deeper and more complex DNN architectures for specifically more improved model accuracy.

But we need to take care of the model tradeoffs with less penalty. The increased demand for ML and embedded applications caused an increase in the research exploration on the DSE of smaller, more efficient and compact DNN architectures.

Also, they can both infer and train faster, as well as transfer faster onto autonomous systems or edge devices. Hence, it is vital to develop a deep architecture with low computational and memory cost with competitive accuracy and model size. The typical and exemplary architectures that had achieved small model size along with competitive model accuracy such as SqueezeNet [1], SquishedNets and SqueezeNext.

This thesis similarly, proposed two DNN architectures to build small, efficient and compact DNN models which would be deployable on autonomous systems. During the architectures development, a great emphasis is laid on the deployability of these models on real-time autonomous hardware platform systems such as BLBX2.0 by NXP and making the following applications possible for autonomous vehicles, drones, robotics, UAV, wireless phones, and all other real-time applications.

1.2 Motivation

Machine Learning is a subset of AI, it is a new modern AI methodology where rather than searching the hard-coded features of an image, a machine is encouraged to learn the image features during the DNN training. This approach is analogous to a human child learning to recognize different objects. Recently, due to the advent of more powerful and computing platforms, the ML for DNN models became relatively easy in contrast to traditional algorithms.

Further, this research aims to contribute to the field of DL by exploring the DSE possibilities of DNNs. Image recognition, classification, tracking, detection, and segmentation are some of the challenging tasks. The state-of-the-art DNN models rivals the human classification accuracy but not necessary in terms of model latency, speed and size. However, real-time deployment becomes a challenge when model size is in Gigabytes or even greater than few Megabytes as big model sizes will have a big memory overhead. The above mentioned image processing or CV tasks can take considerable amount of time. Though the powerful GPUs is one solution but it is not compact and right now it is we do not have the technology feasible to use GPUs

within edge devices. Therefore, we have a growing need to embed these applications on edge devices with memory and power constraints. This manifests a clear need for a minimum DNN model size, model speed and latency time while maintaining the competitive accuracy of the DNN models.

1.3 Problem Formulation

This thesis aims to investigate the field of deep learning by answering the following problem formulations:

- Significance of DSE on CNNs/DNNs.
- The impact of tuning the CNN/ DNN hyperparameters.
- Developing an efficient DNN architecture with minimum tradeoffs from scratch.
- In fact, development of DNN with small model size with competitive model accuracy and speed.
- Visualizing the DNN model accuracy and loss in a real time environment.
- Implementation and deployment of the proposed architectures on real time embedded and autonomous system platform, specifically Bluebox2.0 by NXP.

1.4 Challenges

- Rapid training and testing of DNNs.
- Exploring success ingredients & DSE of DNNs.
- Need a small DNN model size with descent accuracy.
- Need a DNN deployable on autonomous systems.

1.5 Methodology

- Architecture modification.
- Using different datasets and dataset specific training and testing from scratch.
- Fine tuning hyperparameters.
- Implementing different optimizers.
- Finally, testing the proposed architectures on BLUEBOX2.0.

1.6 Contributions

This Master's thesis introduces a number of contributions to different aspects of DSE of DNNs. However, our work is focused on classifying images. This thesis, examines the DSE of DNNs, benchmark the proposed network architecture on CIFAR-10, CIFAR-100 datasets & modify the DNN architecture to reduce the model size while maintaining a competitive level of accuracy. Furthermore, the trained CNN and were deployed on Bluebox 2.0 by NXP using RTMaps remote studio software. The following list mentions the main contributions of this thesis:

- Developing an efficient and flexible DNN architecture with minimum tradeoffs from scratch.
- Achieving better model accuracy.
- Achieving better model size with competitive model speed.
- The impact of changing the hyper parameters of a CNN/DNN such as optimizers, LR, LR schedule, etc.
- Visualize the model accuracy and model loss in real time.
- Comparison of Shallow DNNs versus Deep CNNs.

- Implementation of the proposed architectures on real time embedded system platform i.e. Bluebox2.0 by NXP.
- Training the architecture on GPUs and testing it by deploying it on BLBX2 attaining a competitive model accuracy.
- Two research conference papers.
- Two journal paper publications.

2. LITERATURE REVIEW

2.1 History

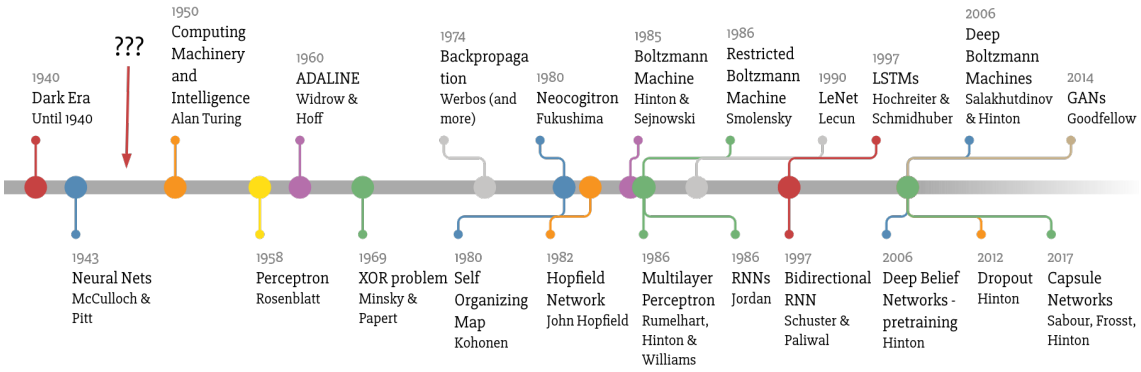


Fig. 2.1. History Timeline of CNNs.

The history of DNN started with a collaborative effort of signal processing and cognitive science community.

Back in 1943, Warren McCulloch, a neuro physiologist and Walter Pitts, a mathematician wrote a paper together [14] on working of neurons in human brain. This paper intended to describe the working of the neurons in the brain, further they also made an electrical circuit to mimic a simple NN. They used a combination of mathematics and algorithms to mimic the though process.

In 1949, Donald Hebb published **The Organization of Behavior**, a research work which showed the facts that neural pathways are the fundamentally essential parts to the human learning. As technology advanced in 1950, a hypothetical NN was simulated, first time. Then, Nathanial Rochester, IBM researcher initiated this NN simulation but he unfortunately failed.

In 1959, B. Widrow and M. Hoff from Stanford developed ADALINE and MADALINE models. Then progressing further a research paper published entitled **Receptive fields of single neurons in the cat's striate cortex** [15] by D. Hubel and T. Wiesel proved to be one of the promising paper in CV for this era of CV technology. In 1959, an equipment was developed that allowed image transformation into grids of numbers that can be easily understood by a machine by R. Kirsch and his colleagues. This equipment used binary language due to their marvellous work, today, we can process digital images in various ways. This gave birth to digital image processing.

In 1960, Henry J. Kelley developed the basics of a continuous backprop algorithm. IN 1962, Stuart Dreyfus developed a simple version of it using the chain rule. But it was still clumsy and inefficient, and until 1985, it would not see any limelight.

In 1962, again B. Widrow & M. Hoff developed a procedure for learning [16]. This procedure examines the weight value before the weight adjusts itself according Weight Change = (Pre-Weight line value) * (Error / (Number of Inputs)). The ideology behind this was that when one active perceptron have a big error rate, weight can adjust coefficient values of it to distribute it across the adjacent perceptrons or the network. Still, if we apply this rule, it will result in an error if the value before the weight is 0 but transversing deep through the net it will eventually correct itself.

In 1963, Lawrence Robert published the research work on **Machine perception of three-dimensional solids** [17] that is widely considered to be one of the predecessors of modern CV era. But the success of the NN didn't last longed enough as the Von Neumann architecture gained popularity. The traditional Von Neumann architecture took over the computing community, and NN research was left behind.

1970s is the period of first AI winter due to lack of fulfillment of promises by AI. It impacted the AI research drastically and funding was withdrawn but some still carried the research without any financial support.

In 1972, Kohonen and Anderson developed a linear auto-associator. They both used mathematics of matrix. They used it to describe the ideas, however they did not realized that they were creating an array of analog ADALINE circuits.

In 1979, it again AI got revived with a first possible CNN by Kunihiko Fukushima, who designed an ANN called Neocognitron. It used a hierarchical, and a multi layered design approach for it. It allowed computer to learn but it was trained with a recurring activation and reinforcement strategy arranged in multiple layers that gained strength over time. The weights within this model were adjusted manually.

In 1982, John Hopfield presented a paper [18] to the National Academy of Sciences, who was from Caltech University. Instead of one way connection between neurons he wanted to create useful machines by using bidirectional lines. It was implemented on NNs and pattern recognition.

With a paper publication named 'Vision: A computational investigation into the human representation and processing of visual information' [19] by David Marr, a British neuro scientist in 1982, NNs again became center of attraction for AI community. As a result of it, there was more investment, and funding. Hence, research in the NN field started with a boost.

In 1986 there was a problem on extending the Widrow-Hoff rule to multiple layers in NNs and then, group of David Rumelhart, a former psychology department member of Stanford came up with a algorithm called back propagation networks. First demonstration of backprop was provided in 1989 by a young French scientist Yann LeCun of Bell labs. He put together backprop with CNN onto a handwritten digits and the computer was able to read it for checks.

1985-1990s represented the break of second AI winter due to the over investments and exaggeration of the potential of the AI at that time. It lead to its downfall and also, due to the better SVM response. SVM was preferred to other AI algorithms and it was performing better in comparison to NNs. But some scientist still continued, without worrying about the much about the downfall and finally, it (NNs) got back on track in 1997.

In 1997, Sepp Hochreiter and Juergen Schmidhuber developed the LSTM algorithm. Also, the SVMs or support vector machine algorithm was devloped by Dana Cortes and Vladimir Vapnik. Then, Jitendra Malik and Jianbo Shi, student at Berkeley professor released a paper [22] to describe his attempts to tackle perceptual grouping.

Then in 1999, a new revolution started with development of GPUs and CPUs became faster too. Both SVMs and NNs started completing with each other, in contrast to SVMs, NNs were slow but more accurate in terms of model results. But around 2000, a new problem called 'vanishing gradient' emerged and this was imminent only within gradient based learning algorithms or methods.

In 2001, Paula Viola and Michael Jones created their first face detection real-time framework that worked in real-time [23]. It was not a DL based algorithm but while processing images, it learned which features that could help with the class score predictions. In mid 2000s the deep learning domain was gaining popularity.

In 2006, the unsupervised pre-training and deep belief nets was introduced by Hinton. In it, the idea was of unsupervised learning was to train 2 simple layers, freeze all the parameters on top of new layer and train only the parameters for the new layer. And in this manner, a deeper network was being tried to be trained. Around this decade the machine learning truly moved from NNs to deep learning. Also, around this year Pascal VOC was launched and provided a standard dataset for object classification along with dataset annotations and dataset itself.

Around 2009, Pedro Felzenszwalb, David McAllester, and Deva Ramanan developed another feature-based model called the Deformable Part Model [24]. In NIPS workshop on DL discovered did not need NN pre training instead a large dataset would be a bigger advantage.

Then, in 2010 ILSVRC was started. It refers to ImageNet Large Scale Visual Recognition Competition. During the annual competition every year CNN achieved a new milestone. AI researcher and Stanford Professor Fei-Fei Li made a huge contribution by building the large dataset of ImageNet.

2012 was a crucial year for DL. Certain AI based pattern-recognition algorithms achieve human-level performance on certain tasks. A. Krizhevsky, I. Sutskever, and G. Hinton entered the ILSVRC competition in 2012 and submitted a CNN model that had laid the foundation for CNNs with a existing error rate to 16%, a model named AlexNet became a state of the art model of deep learning. The model most importantly was trained on GPU, lead to rapid and training on a huge dataset. Further, they also introduced the concept of dropout layer.

With AlexNet, there was boom in the deep learning community. In 2013, M. Zeiler and R. Fergus developed another winning CNN model called ZFNet [26] for the annual ILSVRC. ZFNet was an improvement on the top of AlexNet which was basically done by tweaking the hyperparameters of the architecture.

ILSVRC's 2014 winner was a CNN from Google called GoogLeNet developed by C. Szegedy. In this model , the main contribution of it was the development of a module called inception module which dramatically reduced the number of parameters in the CNN. And also, another CNN model called the VGGNet [1], made a contribution by showing that the depth of the network is an important factor for better CNN model performance.

2015 ILSVRC was won by MSRA developed by Microsoft researcher Kaiming He, et al. It introduced the concept of ResNet blocks or modules.

In 2016 ILSVRC ResNext which was the first runner up CNN model developed by UC San Diego and Facebook AI Research. It improved the structure of ResNet it was the main ideology behind it.

After 2016 ILSVRC handed over the torch of deep learning annual competition to the Kaggle. Most of the winners of the Kaggle used CNN or DNN methodology.

Some of the state of the art CNN models of the new CNN era are MobileNet model family, SqueezeNext, SqueezeNet and recently, introduced GPipe which attained 99% model accuracy on CIFAR-10.

2.2 Neural Network

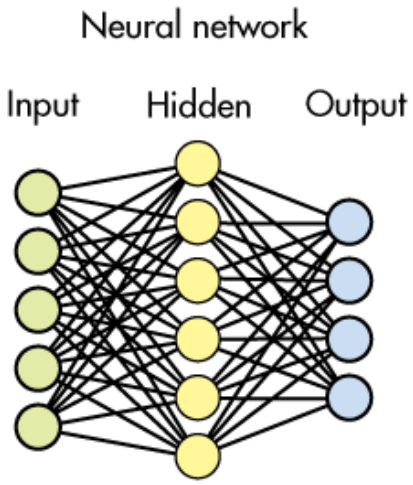


Fig. 2.2. A Simple Neural Network.

Neural Networks are inspired from the human neuron and human brain. Neurons in the human brain are designed to learn patterns and derived casually relations between images and their features. Humans are able to learn because we learn from different kinds of objects with their properties, infer casual relations, using intuitive theories of physics, biology, etc, learn structures, conventions and rules. Neurons within our body interpret sensory data with the help of biological neurons whereas in machine learning we use CNNs/DNNs or ANNs for Computer Vision through machine perception with the help of sensors and cameras. CNNs follows a similar approach to that of neurons and will perform machine perception, clustering (labeling). Data can be in form of numerical data images, sound, text, and video stream. It will help to cluster data and perform image classification. A layer in NN is a collection of stocked neurons which are further processed through activation function. A NN with more than one hidden layer is generally called a DNN.

2.3 Convolutional Neural Network

CNN is a subset of NN domain which is widely used for CV or machine learning applications. It consists of hidden layers which usually contains convolutional layers, pooling layers, fully connected layers, and normalization layers. It has successful implementations of different tasks such as identifying faces [70], objects recognition, detection and traffic signs [3, 41], robotics CV applications and self driving cars.

A CNN example is shown in Fig. 2.3 which takes an image as an input, assign weights and biases to various aspects of the input. Then, some data pre-processing is also required in a CNN to attain better model performance. The input is fed into initial convolution layers to extract the low level feature maps then, we perform sub sampling or pooling operation on the obtained feature maps. We keep on repeating this for few layers of convolution but pooling is optional. In between we can run this through some additional activation layers to introduce non-linearity in the network.

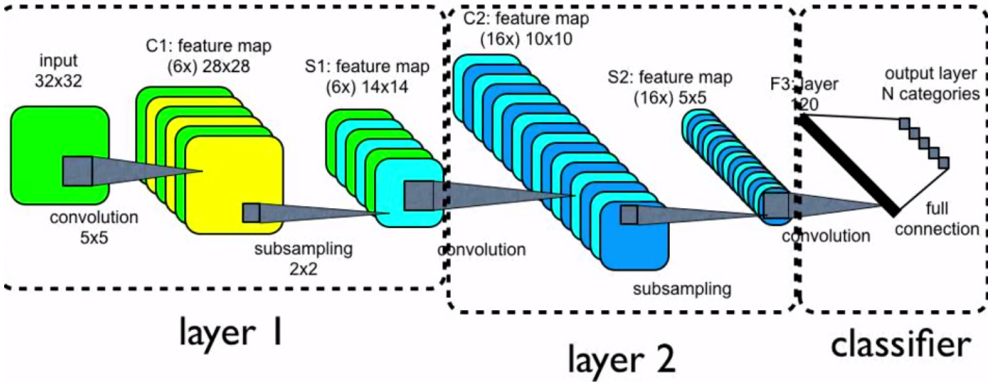


Fig. 2.3. A Two Layered CNN.

CNN architecture is analogous to that of the pattern of neuron connectivity as present in the human brain. Each and every individual neurons respond to a stimuli only in a restricted visual field region that is known as the **Receptive Field**. A collection of them overlap with each other to cover the entire visual area and perform the image processing tasks.

2.4 Classification

Classification in regard to a DNN refers to classifying the images based on the class probabilities. If it is a supervised learning and we perform classification on it that means we have labelled data, it will be easy to classify. Types of classifications are explained below:

- To detect pedestrians, objects, faces and facial expressions [23], detecting audio, voices, music, transcribe speech to text and recognize sentiment in voices.
- Identifying images/objects/gestures in a images or a video stream.

2.5 Clustering

CNN can perform well on labelled data but for unlabelled data clustering is performed. It is a ML technique that would not require labelled datasets but this technique groups data according to the similarities. Unsupervised learning [50] usually use this method for machine learning. An examples of clustering algorithms is K-means.

2.6 Neural Network Elements

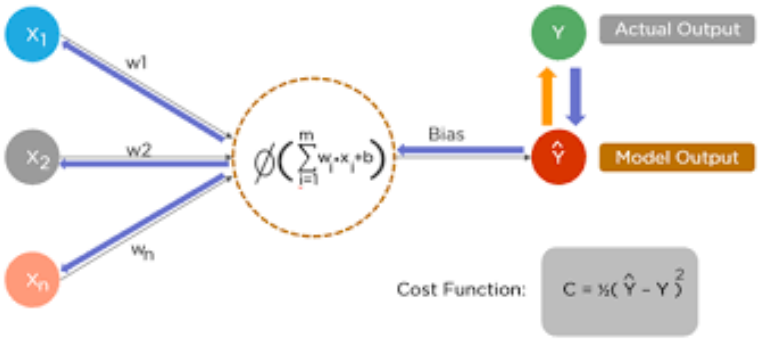


Fig. 2.4. NN Structure Analogous to Biological Neuron.

With NNs the machine leaning era started giving AI a new life. DL is a domain in which we stack multiple NN layers together. The NN used for machine learning consists of nodes (input and outputs), weights, biases, activation function and a feed back mechanism. Inputs and output nodes are responsible for all the computation which is analogous to neuron pattern inside the human brain. Neurons will be triggered or fired when it encounters a stimuli whereas in NNs in ML this happens with the help of input fed into an activation function.

Summation of the input node terms and their respective weights occurs within a place which is also called an activation function. This nodes will further determine whether to trigger a signal or not. If the signal is triggered we can the signal is being activated. Each output of the a NN layer is simultaneously the output of the another subsequent NN layer. In order to adjust the error rate within the NN we use add a bias term to compensate for the error during the NN operation.

2.7 Key Concepts of DNNs

When we have more than one hidden we refer to those as DNNs. The traditional NNs [15, 17, 18, 22] were shallow, composed of one input, one hidden and one output layer. In it, convolutions are responsible to extract set of distinct features for each NN layer for the output of the previous layer fed into the current layer during the DNN model training. The training is often done with the help of powerful GPUs and then the testing can be either done on GPUs, CPUs or real-time embedded systems. The convolutions in the initial layers will learn low features then, mid-level features would be learnt in the middle layers and last layers will learn high features and recombine these features in a FC layer to output the class probabilities. This is called feature hierarchy, and it is a hierarchy of increasing complexity [54, 55] and abstraction.

The concepts of DNNs include data clustering, and classification. DNNs can perform without human intervention for feature extraction from the layers within a DNN model but it is not like most traditional ML or AI algorithms. In a DNN, when

training is done with an unlabeled dataset each input node layer learns features by reconstructing the input from the samples, repeatedly. DNN will try to minimize the error difference between the model prediction and the class probability distribution of the ground truth input. First we perform feed forward propagation and then, backprop is performed in order to correct the error between predicted and ground truth.

A DNN trained on labeled data can then be implemented or deployed to unstructured data combining with the unsupervised learning method when it expands its horizon to more data because most of the real word data is unorganized or unlabelled. For better model performance we need more data or we can use data augmentation. For instance, a poor algorithm trained on lots of large dataset can outperform good algorithms but trained on less amount of data. In the end for output nodes, a DNN can use a logistic, or softmax, classifier or a FC layer such that it assigns the class probabilities and give us the class scores for each of the classes for the input. We also introduce come across a term called babysitting of a CNN which refers to monitor or observe the losses and accuracies of the dataset during training, validation and testing to ensure that our model is not over generalizing or overfitting or even going through underfitting mostly, due to lack or shortage of data.

2.8 Feed Forward Networks

CNN is a methodology which is intended to use for image classification tasks with a human like accuracy. In the beginning of a CNN, it is the place where our CNN model's weights are initialized, default values of hyper parameters are set, features learnt within the convolution layers and in the end, when the parameters or features learnt are capable enough of producing accurate classifications we retrieve output of our CNN model. This is known as feed forward network and we keep on repeating these steps which are similar to each other.

Each layer accounts towards a class prediction, an error, a small weight update, and finally it slowly learns and extract the feature maps of the important features of the input. A collection of weights with their respective hyper parameters, all together in their beginning or ending state, is called a model. Through a CNN model we are trying to derive and build a relationship between ground-truth labels, and predicted labels from a FC layer of the CNN model. As our model learns features over time it we get better model performance and we train a CNN model once on large datasets and then train on small datasets to get better model performance and use the weights of the large datasets to save time, this is known as transfer learning.

A CNN is just like an human infant who is just born and in a stage of constant ignorance, it evolves eventually as it learns. Initially it would not know what weights and what values of biases will transcend the input weights best to make the correct predictions. It will eventually learn as it transverse though more convolutions and perform better with the help of a error feedback system within CNNs which is known as back propagation.

In brief, during training, input is fed into a feed forward NN. The weights, hyper parameters are to some default values. After learning suitable weights and extracting high end features of the input, it provides us with class scores. The ground truth refers to the input images from the specified classes and predicted classes is the prediction made by the network on the basis of learnt featuteres to output the predicted classes and their probabilities.

input * weight = prediction by a CNN

ground truth - prediction by a CNN = error

Difference between the prediction of a CNN and the ground truth of the input class is called its error. A CNN measures this error, and walks the error back over its model, adjusting weights to compensate the error with the help of an approach called backprop.

error * weight's contribution to error = adjustment

All of this as explained above, happens in a repetitive manner in a CNN. Nowadays, training is done on a GPU and then testing is performed on a real time embedded system or CPUs or a hardware with limited resources, which is feasible due to some the advent of macro architectures.

2.9 CNNs over Feed-Forward NNs

An image is a matrix of pixel values for a grayscale image it is between 0-255 or for a binary image it is 0 or 1. We usually fed a image matrix and in the end, it is flatten with the help of a FC layer into vector and feed it to a multi-level perceptron for image classification. This is generally referred to as feed forward NNs but here we do not adjust our weight according to a error within a CNN and no weight adjustment is done, no backprop.

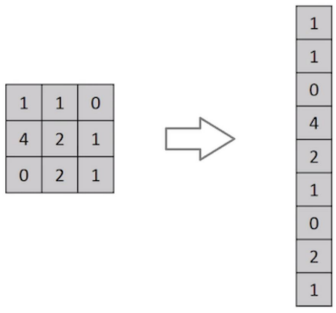


Fig. 2.5. 3x3 Image Matrix into a 9x1 Vector.

A CNN with the application of convolutions it is able to successfully capture the spatial and temporal dependencies in an image. The architecture performs a better due to the reduction in the number of parameters involved and reusability of weights. In a CNN, it can be trained to understand the feature of the image better while having a good class prediction score with a low parameter count in order to deploy it on real time systems or autonomous systems.

2.10 Gradient Descent

Gradient Descent is a commonly used optimization function [53] that adjusts weights according to the error caused within a CNN or DNN. Slope is another reference for a gradient. As shown in the Fig. 2.6 on a xyz plane, a slope represents relationship between two variables in accordance with each other. In the DNN/CNN scenario, the scope of slope defines a relationship between the error of a CNN and an individual weight or in other words, it defines does the error changes as the weight is adjusted.

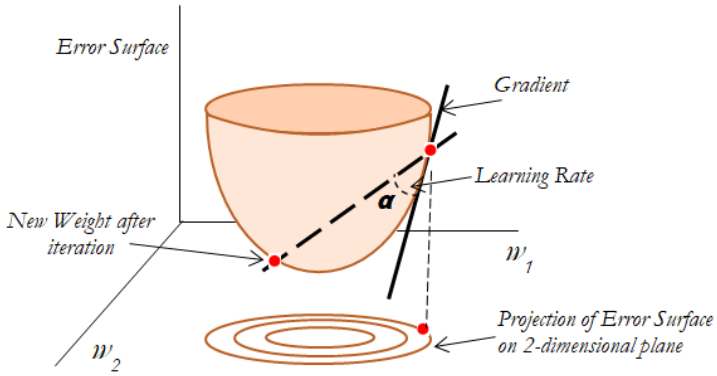


Fig. 2.6. Gradient Descent

As a CNN learns as it runs deep, it will slowly learn and adjusting its weights so that it can map features, accurately. Here we denote the relationship between derivative of the error of a CNN and each of those weights as: dE/dw that measures a small degree of weight change that subsequently causes a small error change. Each weight accounts towards the class scores in a DNN that involves many transforms. The weight learned from the features passes through activations and sums over several other layers. We use a mathematics rule of the calculus, chain rule, in order to travel back through the activations, outputs of the network and finally, arrive at a point where the output weight, and its relationship to the overall network error is significant. The chain rule in calculus states that:

$$\frac{dz}{dx} = \frac{dz}{dy}.\frac{dy}{dx} \tag{2.1}$$

The relationship between the error of the overall CNN and an individual weight in a feed forward network will be same as of backprop chain rule:

$$\frac{dError}{dWeight} = \frac{dError}{dActivation} * \frac{dActivation}{dWeight} \tag{2.2}$$

We have two variables Error and Weight above mediated by a third variable known as Activation, through which weight is being passed. We calculate a Weight change affecting a Error change in the network. But, before this we calculate how a Activation change affects a Error change firstly, and how a change in weight affects a Activation change.

The essence of learning in deep learning is just mainly the a CNN/DNN model weight adjustment in corresponding to the error it produces and we repeat this until we can not reduce the error anymore.

2.11 Process of Training a CNN

Overall training process of a CNN is summarized [26, 33] as below:

- Input image is fed into a CNN which can be a 32x32 (CIFAR-10 OR CIFAR-100) or 224x224 (ImageNet). Then, we initialize all the CNN network, kernels, batch norm and values of the weights with some random values or using initialization functions.

- Forward propagation approach is being followed now in which we perform convolution, activation and pooling operations, finally finally the feature map into a FC layer and output will be class probabilities.

- To let our CNN learn better and make better predictions we calculate the total error of it at the output layer.

$$TotalError = \Sigma \frac{((targetclass\ probability)\ -(output\ probability))^2}{2} \tag{2.3}$$

- Backprop algorithm is implemented to calculate the errors of the gradients with respect to weights of the network and then GD will be used for all weights updation and hyper parameter values to minimize the error of the output.

  - Weights in proportion to their respective total error contribution will be adjusted.

  - Due to weight adjustment the network has learnt to classify better and will be able to generalize in a good way avoiding overfitting.

  - Only weights will be updated and no other hyper parameters such as kernels, ksizes of CNN architecture will be updated during the training of CNN.

- Th process from the above steps 2-4 will be repeated with the training dataset.

When a new input image is fed into a CNN, the network would follow the forward propagation, then backprop for error correction, and finally, providing output for the class probabilities. If we have large dataset our CNN would be able to generalize better and we can add any desired number of convolutions, activation and pooling for our custom CNN architecture but keeping in mind the application constraints for the model performance.

2.12 Visualizing Convolutional Neural Networks

Visualizing is another key part for monitoring a CNN's performance. There are various tools or methods which can implemented but here we are illustrating the concept of it using digit recognition visualization developed by Adam Harley on MNIST dataset for handwritten characters. Please refer [89] for detailed reading about it. Visualization of a CNN is critical as we can observe how well is our CNN model is performing and is there any need to introduce more data, perform more parameter tuning or architecture modification to avoid the overfitting and reaching a saturation point in the network where it cannot learn any further.

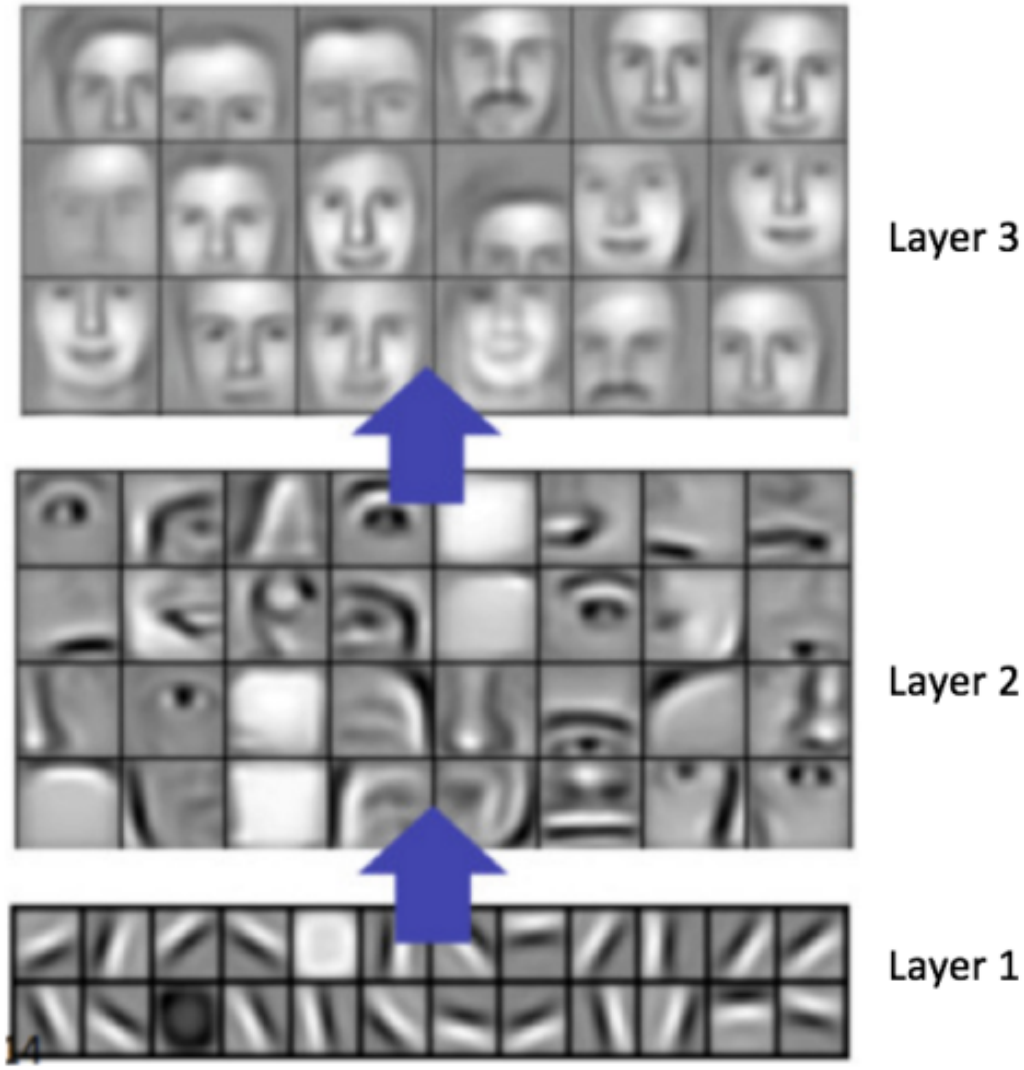


Fig. 2.7. Learned Features from a Convolutional Deep Belief Network.

CNN learn feature maps from the convolution layers. In Fig. 2.7 convolutional deep belief network is used to detect raw pixels edges in the initial first layer. Further, in the second layer the edges learnt are used for simple shapes detection, and then in the third layer these learnt simple shapes are used for further learning the higher-level feature maps that are facial shapes. Fig. 2.7 is used for the illustrating the convolutional deep belief network used for learning face features.

We will see the working of visualization of a CNN network. Fig. 2.8 illustrates the visualization of a CNN for detecting handwritten digits. A 32 x 32 input image is fed into the convolutional deep belief network. The first convolution is formed by six 5x5 convolution with stride value 1.

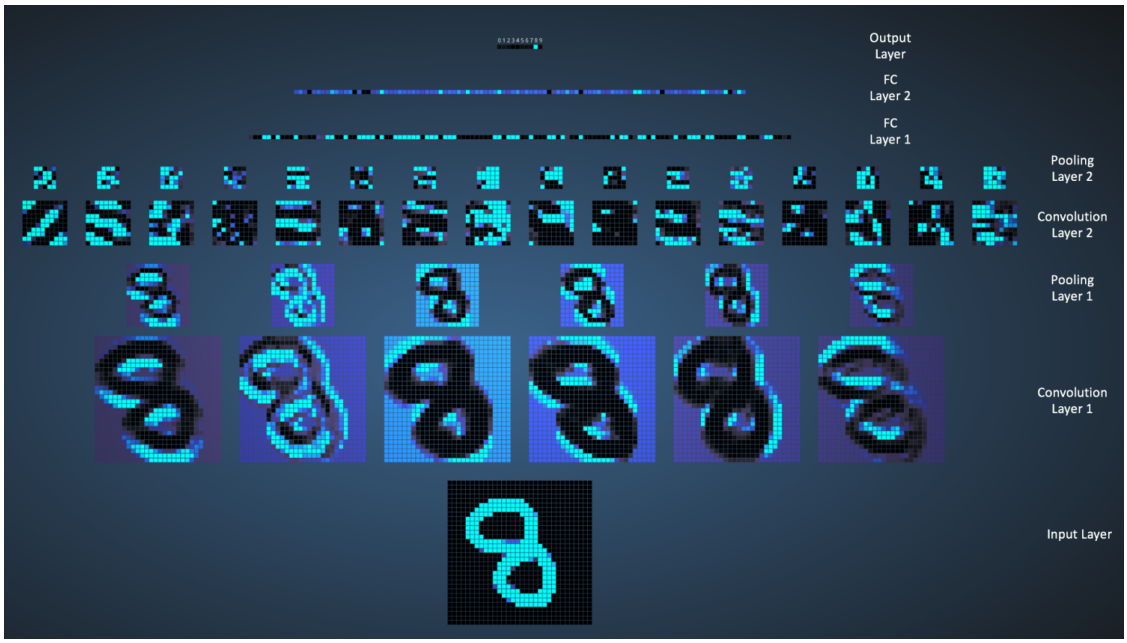


Fig. 2.8. Visualizing a CNN Trained on Handwritten Digits.

We see there are six different layers in Fig. 2.8 which learns features from the input '8' image. Convolution is followed by pooling layer. Convolution is used to extract features and pooling is used for down sampling and in Fig. 2.8 a 2x2 max pooling is used with a stride value 2. In Fig. 2.9, we will observe that the brightest 2x2 grid will be fed in to the pooling layer and the figure illustrates the visualization of pooling operation.

The first pooling layer is followed by 5x5 sixteen convolutions with stride value of 1 that extract features from the input image. This is further, followed by second pooling layer that perform 2x2 max pooling with a value of stride 2. The first FC layer contains 120 neurons, 100 neurons in second FC layer, and finally, the third layer contains 10 neurons.

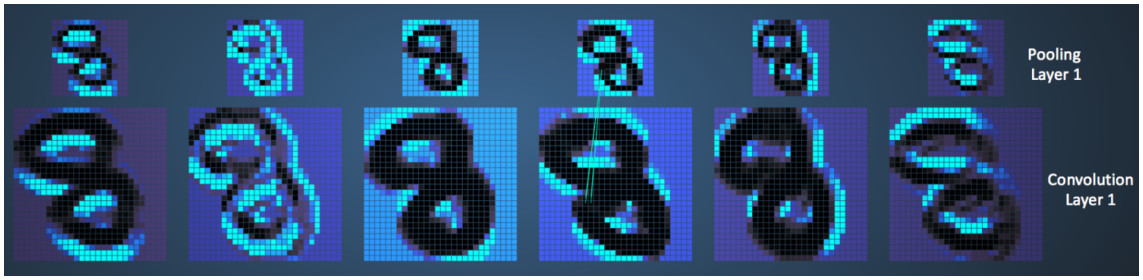


Fig. 2.9. Visualizing the Pooling Operation.

Fig. 2.10, illustrates the visualization of fully connected layer. In the output layer, all the 10 nodes are connected in second fully connected layer, to all the 100 nodes. The bright node or grid in the output layer with fully connected layer corresponds to 8. The output layer classifies the input of handwritten digits.

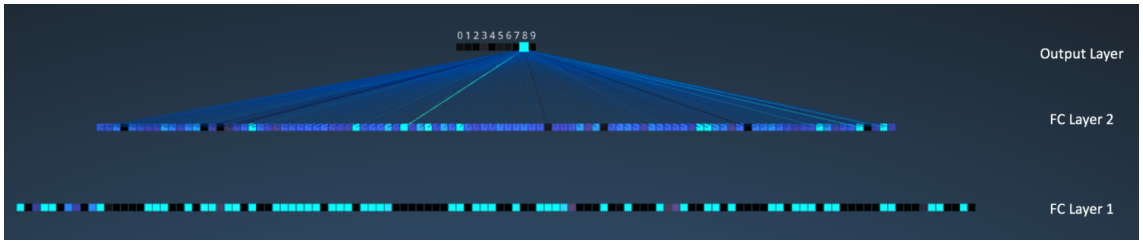


Fig. 2.10. Visualizing the Fully Connected Layers.

3. BACKGROUND

This chapter discusses the basic CNN terminology and methodology behind it. Also, various background related architectures which laid the foundation in developing the new architectures that are High Performance SqueezeNext and Shallow SqueezeNext architectures were reviewed. It also discusses the other techniques and methodologies which were helpful in exploring further, and diving deep into the DSE of DNNs.

3.1 CNNs Elements

In DL we across two terms DNN or CNN. Both are actually related to each other. CNN is a subset of DNN. When DNNs are specially used for image classification and analyzing videos purpose it is generally considered in CNN domain. CNN takes a video or a image input for classification by learning features within the CNN network. In the Fig 3.1, a CNN structure is shown, consisting of 6 layers or 3 basic CNN elemental layers (2 Convolutions, 2 Pooling and 2 FC layers) is shown below. The four main elements of a CNN are essential part of any DNN, shown in Fig. 3.12, which is explained below:

- Convolution layer (Conv).
- Activation or Non Linearity layer.
- Pooling layer.
- FC Layer.

The above elemental layers are the basic building blocks of all CNN architectures or DNN models. It is really crucial to be aware of the knowledge behind it and understand them. The increase of depth and width multipliers of a CNN will increase

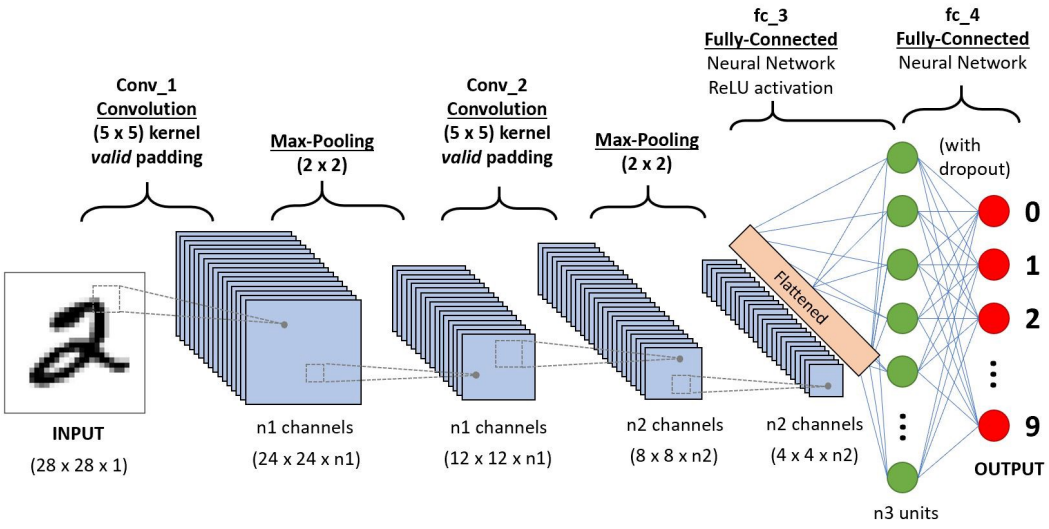


Fig. 3.1. Illustration of a CNN Structure.

the number of parameters within the CNN model due to increase in the number of these four stacked layers basically deep within the model. The intuition behind each of four layers is discussed in detail below and further, insights from these layers will be helpful in architecture development and modification in this thesis study.

3.2 Input Image

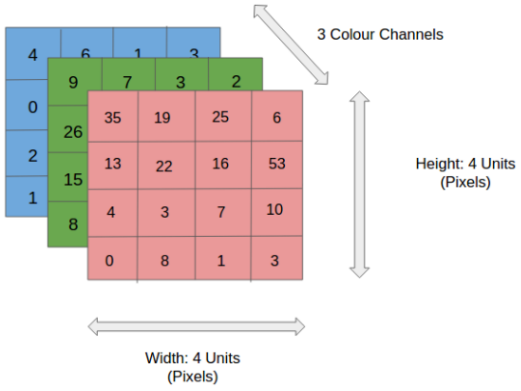


Fig. 3.2. Input RGB Channel Image.

Input image in a CNN can be of any pixel size but for CIFAR-10 dataset it is 32x32 input with 3 channel input, in regard to this study. In that datasets we have 32x32 images of 10 classes that are dogs, cats, ships, planes, automotive , etc. Also, the number of ways in which we can input the color planes or image matrices channels as an input to CNN will be such as grayscale, RGB, HSV, CMYK, etc. The 3 channel RGB matrices are shown separated by its three color planes in Fig. 3.2 with a height and width of 4 units each. The motive of the convolution is to reduce the image channel matrices or down sample them for feature extraction. These features learnt in the convolution would be responsible for class predictions or scores in the end. It is important to design architectures with less parameter count with decent model accuracy.

3.3 Convolution Layer (Kernel)

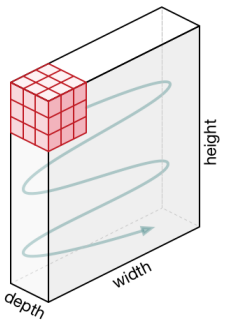
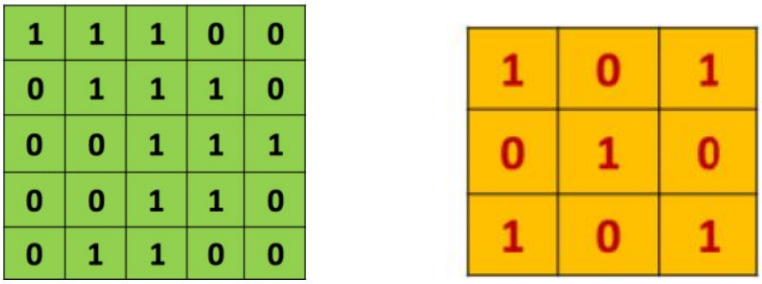


Fig. 3.3. Kernel Movement

Convolution layer is used in order to extract features (low, mid and high level) from the input image. In the beginning during the initial convolution layers it will extract low level features. Mid-level features were obtained in the middle convolutions and lastly, high-level features are retrieved in the last convolutions latter down in the CNN model. It also, preserves the spatial relationship between pixels by learning features. Fig. 3.3 illustrates the movement of a filter or a kernel along a input channel. In Fig. 3.4, a 5x5 matrix (green) is one input channel to be used with a 3x3 kernel or

a filter (Fig. 3.4(b)) for convolution. The stride value used during this convolution operation is used was with value 1. To get a convolved feature we perform a element wise multiplication between the 5x5 (green) and 3x3 (orange) matrices, shown in Fig. 3.4, for each element and then after addition of these multiplication gives out the output of each element of the 3x3 (pink) matrix, shown in Fig. 3.5. After the performing convolution operation, we will get a convolved feature similar to 3x3 pink matrix as illustrated in Fig. 3.5.



(a) 5 X 5 Matrix. (b) 3x3 Matrix.

Fig. 3.4. Input Image with 3x3 Kernel.

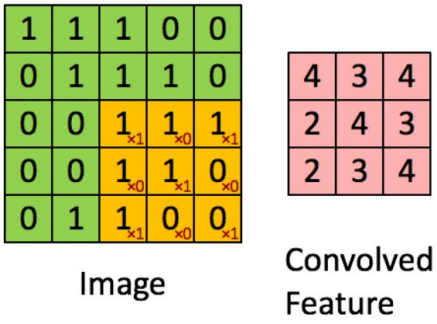


Fig. 3.5. Illustration of Convolved Feature.

In a CNN, the 3x3 orange matrix refers to a kernel/ filter/ feature detector. It is a matrix which we slide over the input image, computing the dot product to output a pink matrix also knows as the convolved feature/ activation map/ feature map. We

can use different types of kernels but usually 1x1, 3x3 and 5x5 are used. More number of kernels will results in more of features extracted. So we will have more features learnt. Hence, our CNN model becomes better and accurate in class predictions or target score.

3.3.1 Convolution Layer Parameters

The size of the feature map or convolved feature (pink matrix) is controlled by three other parameters which we need to decide before the convolution will be performed:

- **Depth** refers to the number of filters intended to be used for the convolution and this further also effect the number of parameters in a CNN.
- **Stride** is the number of pixels that will be a stride value of 1,2 or 3 (usually these stride are used) which kernel slide over the input matrix. When the stride value is 1 kernel moves one pixel, with value 2 or 3 it will move 2 or 3 pixels, respectively. We usually prefer to use stride value 1 (small datasets) or 2 (large datasets).
- **Padding:** is the way to add few extra pixels along the borders of the image or channel for convenience of kernel application over the input. Different types of it are available but zero padding is most widely used. It allows us to control the size of the feature maps.

3.4 Non Linearity (ReLU)

Non linearity or also knows as activation layer is an important element of a CNN. Most commonly used activation function is ReLU but in this study we use ReLU in place and ELU in place. The purpose of this layer is to introduce non-linearity in our CNN, since most of the real-world data would be non-linear. The ReLU operation is shown and could be understood clearly from Fig. 3.7. The output feature map is

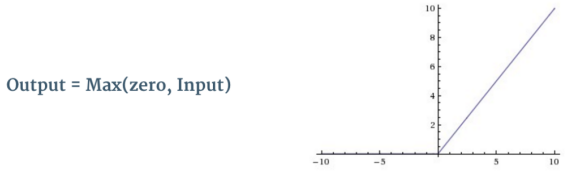


Fig. 3.6. ReLU's Mathematical Function & Graph

called as the rectified feature map in Fig. 3.7. Other non linear functions [53] such as sigmoid, tanh, LeakyReLU, ELU, SELU can also be used. More detailed explanation is provided in chapter 5. The convolution layer performs linear operations basically on the input and it introduces non linearity in a CNN without affecting receptive fields of a CNN.

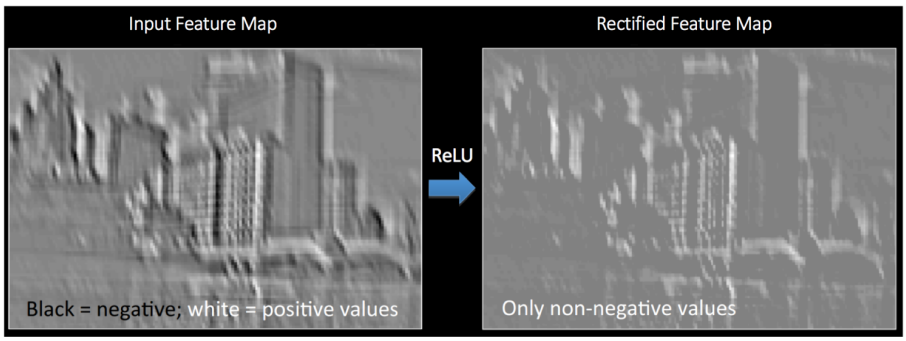


Fig. 3.7. ReLU Operation.

3.5 Pooling layer

Pooling is also known as down sampling/ sub sampling. There are several types of pooling layers such as max, average, adaptive max, adaptive average pooling, etc. Most often, we use max pooling layer to reduce the dimensionality of individual feature maps. It reduces the spatial dimension but at the same time retains the most important features. Fig. 3,8 illustrates max pooling and average pooling operations. Both have their pros and cons. Then, Fig. 3.9 illustrates how max pooling operation

is implemented after a convolution and ReLU on a rectified feature map. Here in both figure we slide 2x2 matrix by 2 strides and take the maximum value for max pooling or average for average pooling operation.

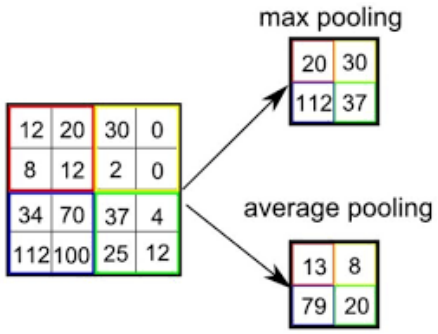


Fig. 3.8. Pooling Operation (Max and Average Pooling).

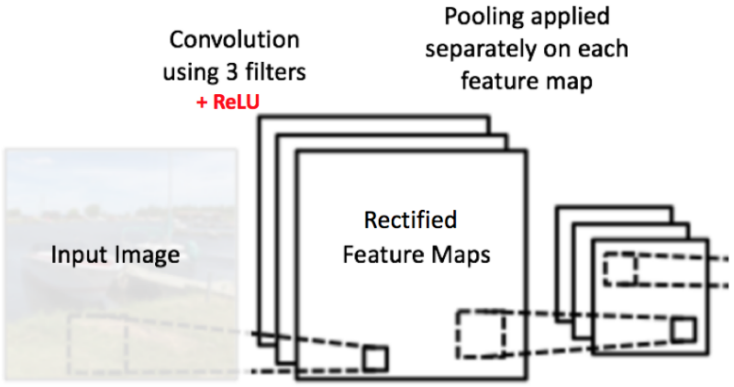


Fig. 3.9. Pooling Applied to Rectified Feature Maps.

In Fig. 3.10, max and average pooling operation is applied separately to a rectified feature map. Here, in this image we see two pooling outputs. Pooling reduces the number of parameters and computations in a CNN and also cope up with the overfitting problem. Overfitting problem refers to when a CNN model over tunes the weight according to training examples such that it is not able to generalize well for the validation and testing images.

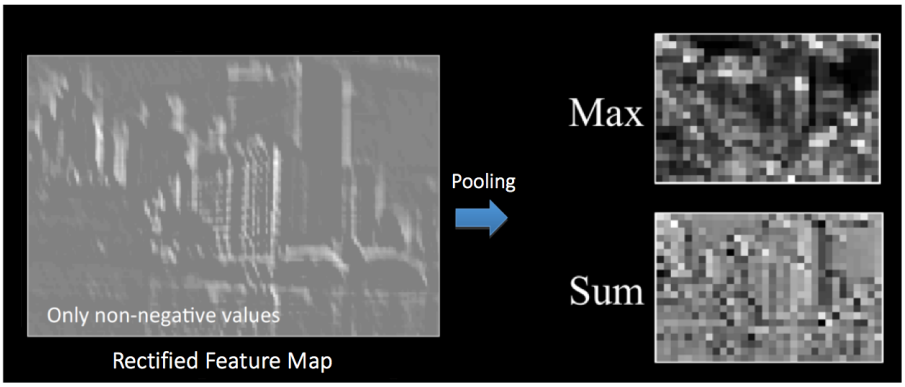


Fig. 3.10. Pooling Operation Illustration.

It is observed when there is a big difference between training and testing model accuracy curves during the model visualization. It makes the network invariant to small distortions, translations, and transformations in an input image. This is very useful as we can detect objects in an image it would not matter where these objects are located. Further, the output of the pooling layer in the end is fed as an input to the FC Layer.

3.6 Fully Connected Layer

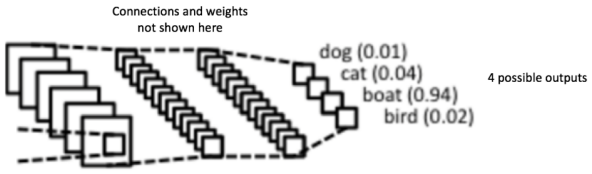


Fig. 3.11. A FC Layer.

FC layer is a traditional multi-layer perceptron. This layer uses a softmax function in the output layer as an activation. Here 'Fully Connected' means that each and every input node in the previous layer is connected to each and every output node on the next layer. In the end of a CNN model, the output from the convolution and

pooling layers extracts high-level features of the input. In order to use these features for classifying the input into different target classes or providing class scores on the basis of the training dataset. For example, in Fig. 3.11 the image classification task is to perform has four possible outputs such as dog, cat, boat and bird.

Also, it is usually an easy and more effective of learning non-linear combinations of these features in the end. We observe we get the sum of output probabilities from the FC Layer in total is equal to 1.00. The softmax activation function in the FC layer ensures this and it takes a vector of real-valued scores and squashes it to a vector between zero and one values that sum equal to 1.00 total probability.

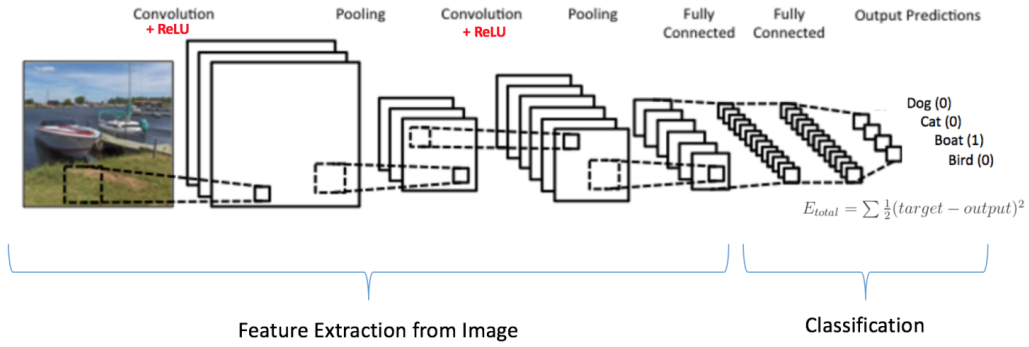


Fig. 3.12. Training a CNN Classifier.

3.7 Classifier

From above we see FC layer is majorly responsible for classification inf a CNN. The convolution and pooling layers are responsible for feature extraction from the input channels. In Fig. 3.12, example of training a CNN with the input image of a boat represents a classifier example. The boat class target probability is 1 and 0 for other classes (dog, cat and bird). After learning the features in the CNN model we are able to get a 1 probability for boat class which showes that CNN is successfully able to classify the boat image after learning its features.

3.8 Parameter Calculation in CNNs

The concept of how a CNN learn is also an essential element for this study. A CNN is trying to learn features or feature maps with the help of convolution(s) or kernel(s) using backprop. In a CNN, the parameters are usually learnt in convolution and fully connected layers but not in pooling or activation layers.

3.8.1 Calculating the Number of Parameters in CNNs

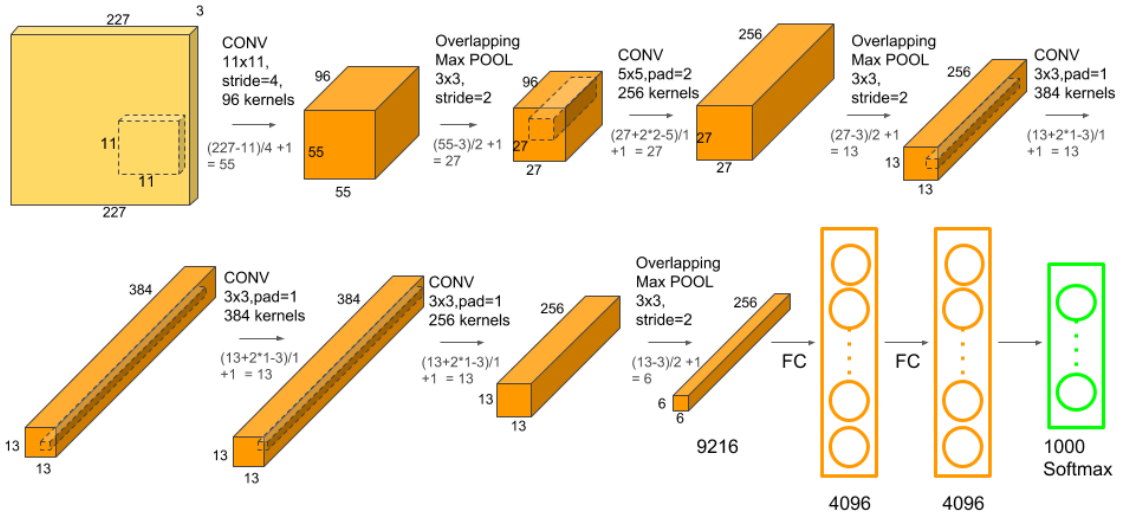


Fig. 3.13. CNN Parameters Calculation Illustration.

The formula (3.1) is used to calculate the activation shape or the number of output features, that is denoted by n_{out}, the number of input features is denoted by n_{in}, p refers to convolution padding size, k corresponds to convolution filter size, and convolution stride size is represented by s. In the last, we add one for the bias term. Now let us see where CNN exactly learns at which layers and how.

$$n_{out} = [\frac{n_{in} + 2p - k}{s}] + 1 \tag{3.1}$$

- **Input layer** has nothing to learn, so no parameters are learnt here in CNN. It just feds input image to CNN.

- **Convolution layer** is a place where CNN extract features and learns. Here we will have weights. Parameter calculation for the parameters learnt is multiplication of the shape of width m, height n, kernels k and adding a bias term for each kernel. Formula for parameters learnt in convolution layer: ((m * n)+1)*k) or in other words, ((shape of width of the filter * shape of height of the filter + 1) * number of filters).

- **Pooling layer** again, got nothing to learn or no learnable parameters. The aim of this layer is to provide down sampling of the input.

- **FC Layer** has certainly alot of learnable parameters. Every input node is connected to every output node here in this layer. The parameter calculation includes the product of the number of nodes in the current layer and the number of nodes in the previous layer. Therefore, we can say formula for this is: ((current layer n * previous layer n) +1).

Table 3.1.
CNN Parameters Calculation Example

Layer name	**Activation shape**	**Activation size**	**Number of parameters**
Input	(32,32,3)	3,072	0
Conv1 (f=5, s=1)	(28,28,8)	6.272	208
Pool1	(14,14,8)	1,568	0
Conv2 (f=5, s=1)	(10,10,16)	1,600	416
Pool2	(5,5,16)	400	0
FC3	(120,1)	120	48,001
FC4	(84,1)	84	10,081
Softmax	(10,1)	10	841

3.9 Related Architectures

3.9.1 SqueezeNet Architecture

This section reviews the SqueezeNet architecture [5] which comprises of fire modules, Relu activation, max and average pool layers, softmax activation, and kaiming uniform initialization. Fire module is the backbone of this architecture, comprising of a squeeze layer, s2 (1x1) and two expand layers, e1 (1x1) and e3 (3x3). The three following design strategies are employed to construct the baseline squeezenet architecture:

- Strategy1: Replace 3x3 filters with 1x1 filters.
- Strategy2: Decrease the number of input channels to 3x3 filters.
- Strategy3: Down sample late in the network.

Fire modules greatly reduce the number of parameters as compared to the state of the art VGG architectures. The VGG architecture with 385MB model size was reduced down to 2.5 MB of squeezenet baseline's model size with decent accuracy. The squeezenet architecture has a scope of further improvement with the help of using the following methods such as batch normalization layers [11], element-wise addition skip connections, in-place operations and other types of the optimizers. Fig. 3.14 illustrates the SqueezeNet baseline architecture along with the representation of fire module.

SqueezeNet [5] is the state-of-the-art CNN model with a good model size with a competitive accuracy. It uses 1x1 and 3x3 convolutional kernels. Using 1x1 reduces depth, so, it will further reduces the computation of the 3x3 kernels. It has 50x fewer parameters than AlexNet with a same accuracy for ImageNet. The unique feature of SqueezeNet bseline is a lack of FC layers. It uses an average pooling layer to calculate class scores or target score using kernels instead of a FC layer. This again, have reduced computation & memory requirement by a great amount.

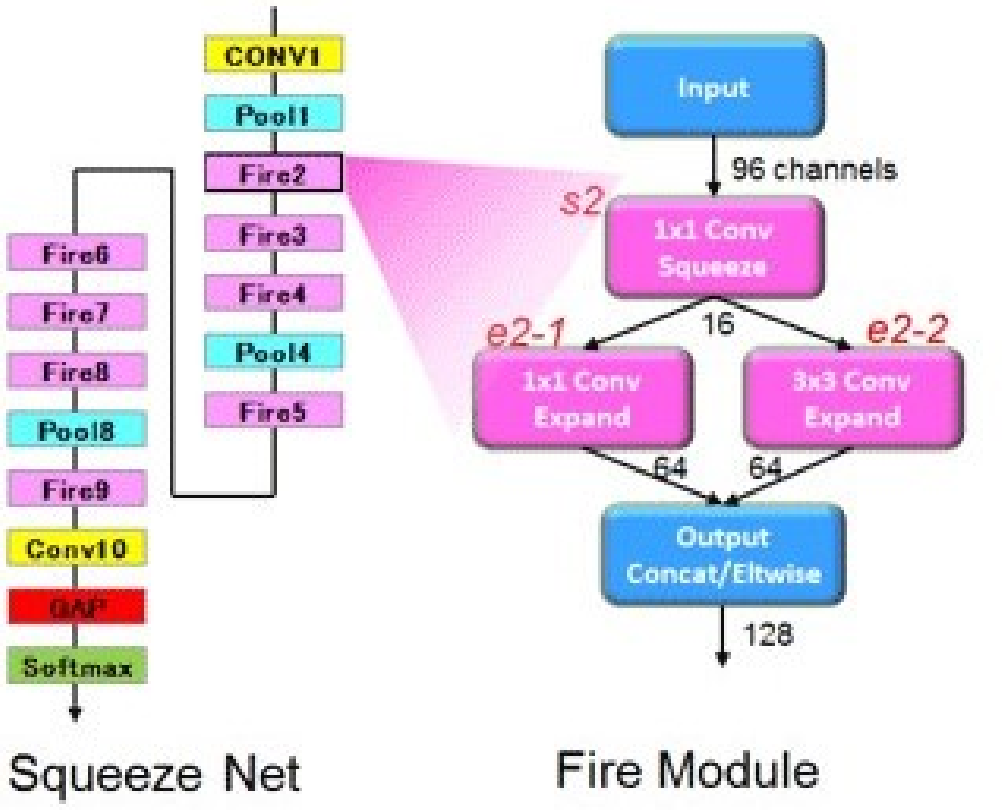


Fig. 3.14. SqueezeNet Baseline Architecture & Fire Module

This makes SqueezeNet best suited for the embedded or autonomous platform and benchmark CNN architecture for it. Then, SqueezeNet v1.1 came into existence in which the number of filters as well as the filter sizes are further reduced. It now, resulted in 2.4x further, less computation than the original SqueezeNet without sacrificing model accuracy which is even better. Inspired by these incredibly small macro architecture of SqueezeNet v1.0 and v1.1, CNN literature review insights, and new methodologies, ultimately, contributes to architecture modifications made within the proposed architectures.

3.9.2 SqueezeNext Architecture

SqueezeNext baseline architecture [6] emerged after the advent of the SqueezeNet architecture. SqueezeNext uses SqueezeNet architecture as a baseline and consists of the following:

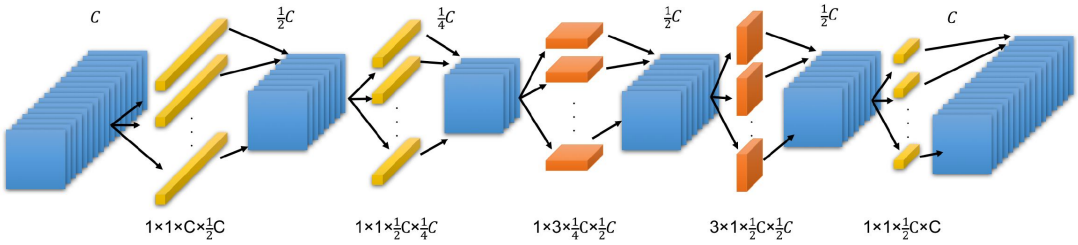


Fig. 3.15. Bottleneck Module in SqueezeNext Baseline Architecture.

- A more aggressive channel reduction by incorporating a two-stage squeeze module, significantly reducing the total number of parameters used with the 3x3 convolutions.
- Separable 3x3 convolutions to further reduce the model size, and remove the additional 1x1 branch after the squeeze module.
- An element-wise addition skip connection similar to that of ResNet architecture [44, 45, 47], this allows it to train a much deeper network without the vanishing gradient problem.

SqueezeNext baseline architecture [6] comprises of bottleneck modules with four stage implementation, batch normalization layers, Relu and Relu (in-place) nonlinear activations, max, and average pool layers, Xavier uniform initialization, a spatial resolution layer and a FC layer in the last with this (6,6,8,1) four stage block configuration. Bottleneck module shown in Fig. 3.15 is the backbone of the SqueezeNext architecture as it significantly reduces the number of parameters without the deterioration of the model accuracy.

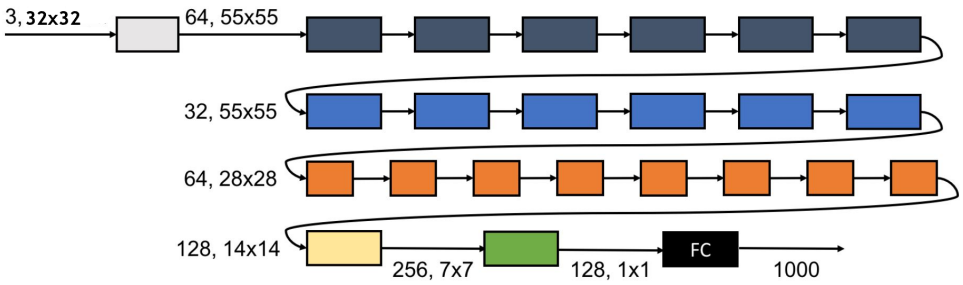


Fig. 3.16. SqueezeNext Baseline Architecture for CIFAR-10.

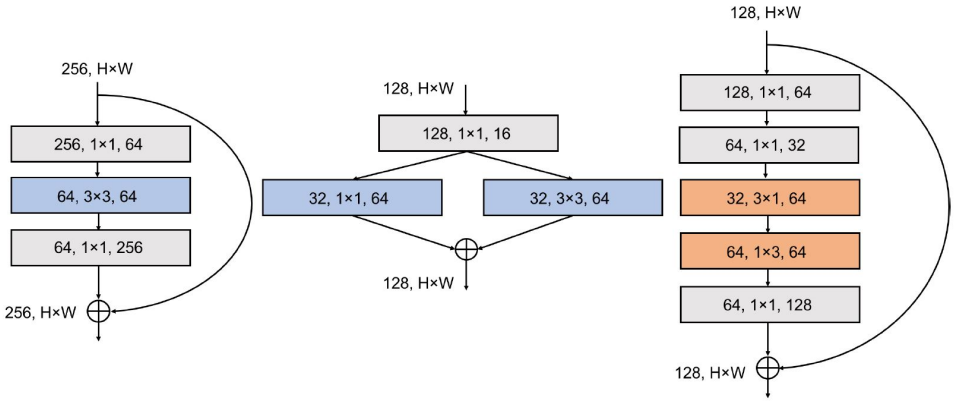


Fig. 3.17. Illustration of Baseline Architecture's Modules of ResNet, SqueezeNet, SqueezeNext.

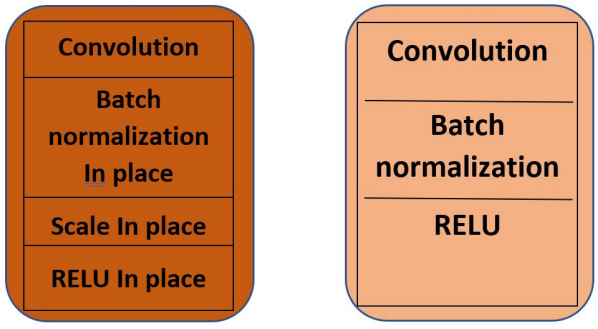


Fig. 3.18. Squeezenext Baseline Basic-Block Module, SqueezeNext's Pytorch Basic-block Module Iuuuyis Version.

In fact, a better model accuracy and size is attained in comparison to squeezenet baseline architecture. The squeezenext baseline (6,6,8,1) architecture configuration shown in Fig. 3.16 illustrates the squeezenext baseline architecture implemented on the CIFAR-10 and CIFAR-100 datasets with input size 32x32 and 3 input channels. This is the input for the first convolution, the white block.

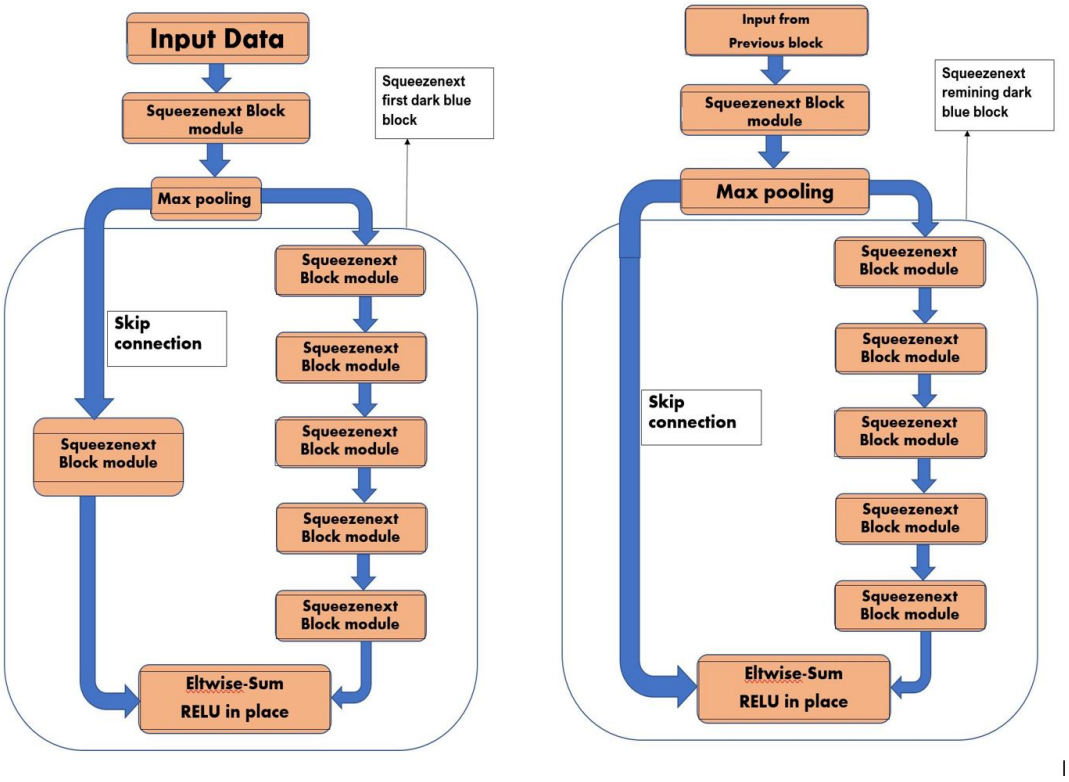


Fig. 3.19. SqueezeNext First Block Structure, SqueezeNext Second Block Structure.

Then after this, the output of the first convolution is the input to a max pooling layer after the first convolution (white block), not shown in Fig. 3.16, but shown in Fig. 3.19. The consecutive different colored blocks that are dark blue, blue, orange and yellow blocks after the first convolution and max-pooling represents the four-stage configuration implementation followed by a green block, representing the spatial resolution layer and the average pooling layer.

Finally, followed by one black block that is a FC layer. The color change in the four stage implementation configuration blocks that are dark blue, blue, orange and yellow blocks depict a change in the input feature map's resolution. The SqueezeNext basic-block structures are shown in Fig. 3.19 are the building blocks of the squeezenext baseline architecture. The SqueezeNext's first block structure comprises of a unique trait that there is only one skip connection is on the left hand side, is used for rest of all the remaining same colored blocks except the first block of each color chain in four stage implementation of the baseline SqueezeNext.

The SqueezeNext's second block structure on the other hand, comprises of a skip connection along with an extra SqueezeNext block module, is used as beginning first block of the same colored chain shown in Fig. 3.16. All squeezenext block modules here refer to baseline block module in Fig. 3.19. SqueezeNext baseline architecture is trained and tested from scratch on the CIFAR-10 dataset with input size 32x32 and 3 input channels with 10 target classes for experiments in this thesis study.

4. HARDWARE & SOFTWARE FRAMEWORK

4.1 Hardware Used

- Intel i9 8th generation processor with 32 GB RAM.
- Required memory for dataset and results: 4GB.
- NVIDIA RTX 2080 Ti GPU.
- NVIDIA GTX 1080 SC GPU.
- ROM 25 GB.
- BlueBox 2.0.

4.2 Bluebox2.0

Fig. 4.1. Bluebox2.0 by NXP.

The NXP BlueBox is a development platform series that provides the required performance, functional safety and automotive reliability for engineers to develop self-driving cars. The latest addition to the series, the BLBX2-xx family, incorporates

the S32V234 automotive vision and sensor fusion processor, the LS2084A embedded computer processor and the S32R27 radar microcontroller. Basic implementation of BLBX2 is explained in this paper [43].

4.2.1 Specifications

- ASIL-B compute, automotive interfaces with vision acceleration.
- ASIL-D subsystem, with dedicated interfaces.
- Automotive I/O, numerous interfaces.
- 12 V /24 V vehicle compatible power input.
- High performance compute with 16 GB DDR4 and 256 GB SSD.
- Ethernet 100M/ 1G/ 10Gbps, SFP+, 8x 100BASE-T1, CAN-FD, FlexRay, 8x cameras.
- HIGHLY OPTIMIZED SENSOR FUSION: Various sensor data streams: radar, vision, LiDAR, V2X.
- S32V234 automotive vision and sensor fusion processor.
- LS2084A embedded compute processor.
- S32R27 radar microcontroller.
- CSE and ARM TrustZone technology.
- ROS Space, Open ROS Space Linux-based system.
- Programmable in linear C, easily customizable, development environment for mainstream vehicles.
- Up to 90,000 DMIPS at < 40 W, complete situational assessment, supporting classification.

The NXP BlueBox platform delivers the performance required to analyze driving assistance systems or environments [41, 42]. It is also a ASIL-B and ASIL-D compliant hardware system. It assess risk factors and then direct the behavior of autonomous car or driver assistance system. BLBX2 is one of the real time embedded development platforms designed for the ADAS system. It is an integrated package for creating autonomous applications such as ADAS systems, driver assistance systems [64, 84]. It is comprises of three independent systems on chip that are S32V234: a vision processor, LS2084A: a compute processor, and S32R274:a radar microcontroller.

Fig. 4.1. shows the actual BLBX2 real time embedded system platform. The toolkit in this is a central processing engine combined with a grid of sensors and other components or sensors, creating a top-to-bottom solution. It is based on NXP semiconductor and APEX processors that are highly parallel computing units with SIMD architecture. It can handle data level parallelism or multi thread or parallel architectures in a good manner. One of the significant requirements of this thesis is to analyze the capability of NXP BLBX2 as an autonomous embedded platform system for real-time autonomous applications.

It can be a potential development real time system hardware for level 5 autonomous cars. The system uses one of the Cortex-A72 layers cape processors out of the 8 processors and an embedded vision chip S32V234 designed around the Cortex-A53 core. NXP has addressed ADAS systems including Level 1 deploying collision warnings, automatic brakes and maintaining a set vehicle distance from others. Level 2 technology implementation of car steering, brake, and accelerate automatically with in limited conditions and constraints. It would not eliminate the need of a human driver here though.

Level 3 autonomous applications is where the driver can completely hand over safety-critical functions in certain situations. Level 5 autonomous cars are going to need a significant amount of memory and computation resources. The challenge here is giving autonomous cars the ability to take over more driving functions, more computation and memory resources with a fail proof system with a complete safety,

security and reliability. One challenge facing autonomous driving developers is proving the safety of the autonomous systems. That is why, we still need new few more years to come up with safe fail proof and further, more advanced and cutting edge high end real time autonomous systems.

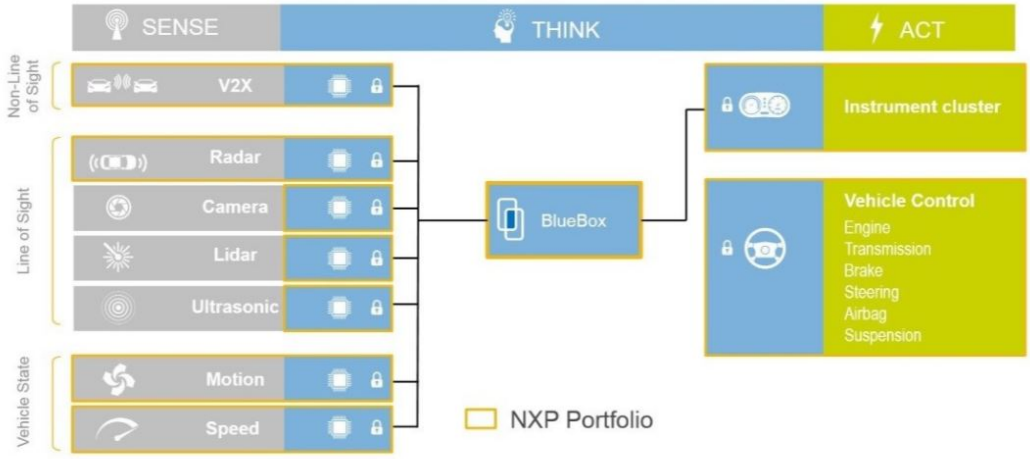


Fig. 4.2. NXP ADAS Real Time Sensor Network, BLBX2.0: Development Platform for ADAS systems.

Hardware Architecture for BLBX2 is shown in Fig. 4.3. The BLBX2 operates on the independent embedded linux OS BSP package for both the S32V and LS2 processors, the S32R runs a bare-metal code that refers to RTOS environment. BlueBox functions as the central computing unit of the system thus, providing the ADAS system to be capable of deploying efficient and better CNN/DNN models.

4.2.2 S32V234 - Vision Processor

The S32V234 Micro processing unit offers an ISP, powerful 3D GPU, dual APEX-2 vision accelerators, automotive-grade reliability, functional safety, and for supporting computation intensive ADAS, NCAP front camera, object detection and recognition, surround view, automotive and industrial image processing, also ML and sensor fusion applications. It is a 2nd generation vision processor family and member of the 32-

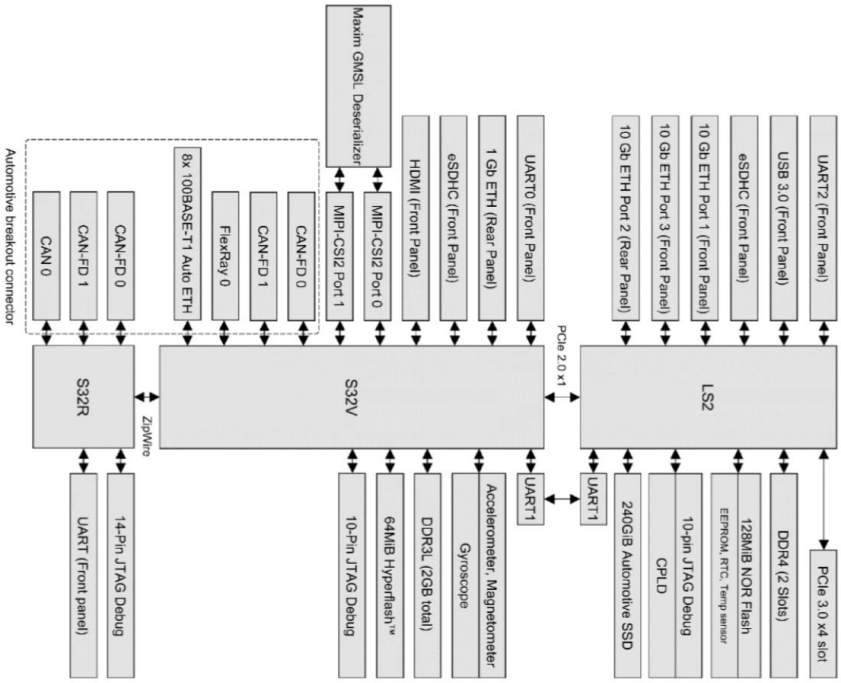


Fig. 4.3. Hardware Architecture for BLBX2 by NXP.

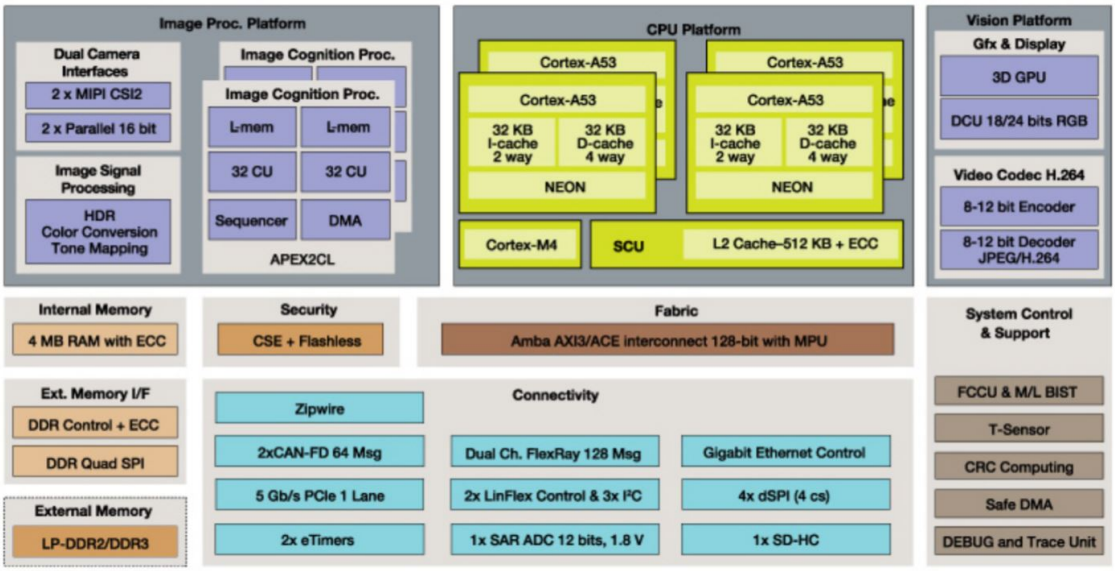


Fig. 4.4. Hardware Architecture for S32V234 by NXP.

bit Arm Cortex-A53 S32V processors and is supported by S32 Design Studio IDE for development of vision applications. Design studio includes a compiler, debugger, Vision SDK, Linux BSP, and graph tools.

It is a vision-based processor designed for computationally intensive application related to the image and video processing or CV applications. The processor comprises of ISP available on all MIPI-CSI camera inputs, providing the functionality of to integrate multiple cameras for image conditioning. It contains APEX-2 vision accelerators and a GPU designed to accelerate CV functions such as object detection, recognition, surround view, ML applications. It also, contains 4 ARM Cortex-A53 cores, and an ARM M4 core designed for embedded related applications. The processor can operates on Linux BSP, Ubuntu 16.04 LTS and NXP vision SDK. The processor boots up from the SD card which is interfaced at the front panel of the BLBX2.0. A comprehensive overview of the S32V234 processor is shown in Fig. 4.4.

4.2.3 LS-2084A

The LS2 processor in the BLBX2 is high-performance computing processor platform. It consists of eight ARM Cortex-A72 cores, 10 Gb Ethernet ports, supports a high total capacity of DDR4 memory, and features a PCIe expansion slot for any additional hardware such as GPUs or FPGAs. All this makes it suitable for applications that demand high performance , and support for multiple concurrent threads with low latency. The hardware architecture for LS2084A is shown in Fig. 4.5, a comprehensive overview of it. The LS2 is also a convenient platform to develop the ARMV8 code and is connected to a Lite-On Automotive Solid State Drive via SATA, to provide large memory size for software installation. It also consists of SD card interface which allows the processor to run: Linux BSP, Ubuntu 16.04 LTS as OS4.3 platform on the bluebox platform.

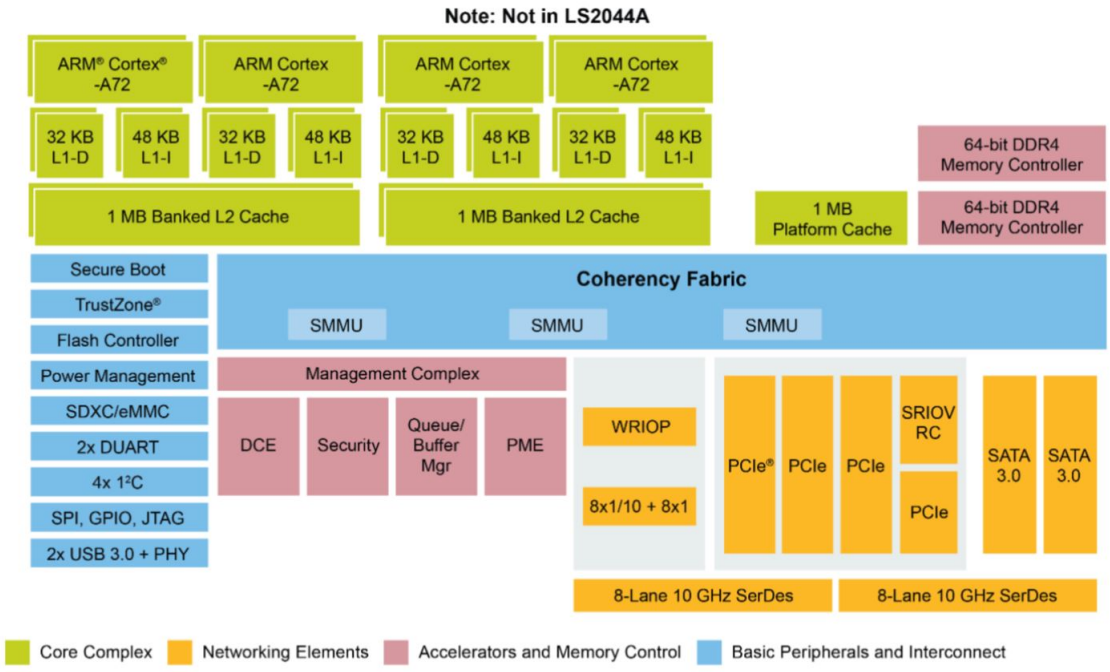


Fig. 4.5. Hardware Architecture for LS2084A by NXP.

For this thesis, the software enablement on the LS2084A and S32V234 SoC is deployed using the Linux BSP. The LS2084A and S32V234 SoC are installed with Ubuntu 16.04 LTS which is a complete, developer-supported system. It contains the complete kernel source code, compilers, toolchains, with ROS kinetic and docker package. The QorIQ LS2 family of processors delivers unmatched performance and integration for smarter, and capable CNN/DNN networks. The eight core QorIQ LS2084A and the four core LS2044A multi core processors offer Arm Cortex-A72 cores with advanced, high-performance data path and network peripheral interfaces required for networking, datacom, wireless infrastructure, military and aerospace applications. The data path architecture combined with a software toolkit provides a higher level of hardware abstraction and makes software development fast and simple.

4.3 Software Used

- Spyder version 3.7.2.
- Pytorch version 1.1.
- RTMaps 4.0.
- Livelossplot (Loss and accuracy visualization).
- Architecture visualization: Netscope.

4.4 Real-Time Multi Sensor Applications (RTMaps)

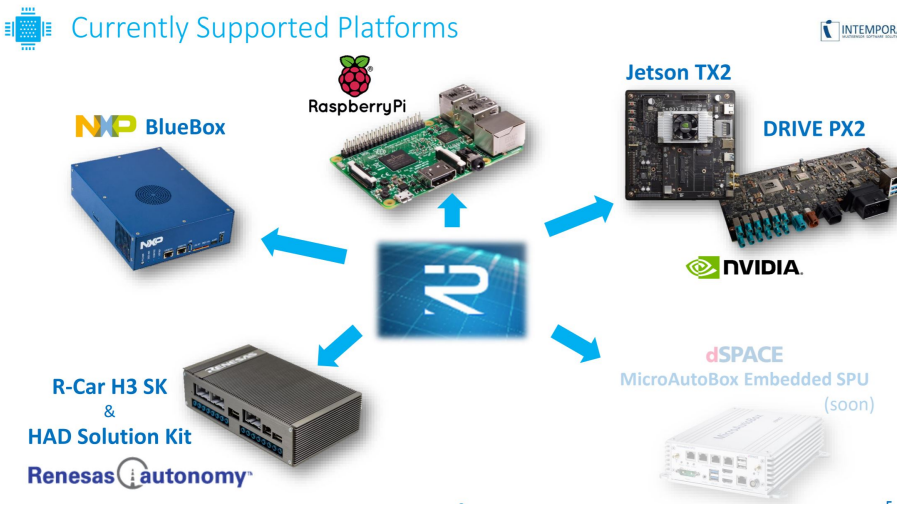


Fig. 4.6. RTMaps Currently Supported Platforms.

RTMaps is an asynchronous high performance platform designed to face and win multisensor challenges and to allow engineers and researchers to take an advantage of an efficient and easy-to-use framework for fast and robust developments. It is a modular toolkit for multimodal applications. ADAS, autonomous vehicles, robotics, UGVs, UAVs, HMI, datalogging. The easiest way to develop, test, validate, benchmark and execute applications is designed for the

development of multimodal based applications [42], thus providing the feature of incorporating multiple sensors such as camera, lidar, radar. It has been tested for processing and fusing the data streams in the real-time or even in the post-processing scenarios [63]. The software architecture consists of several independent modules that can be used for different situation and circumstance.

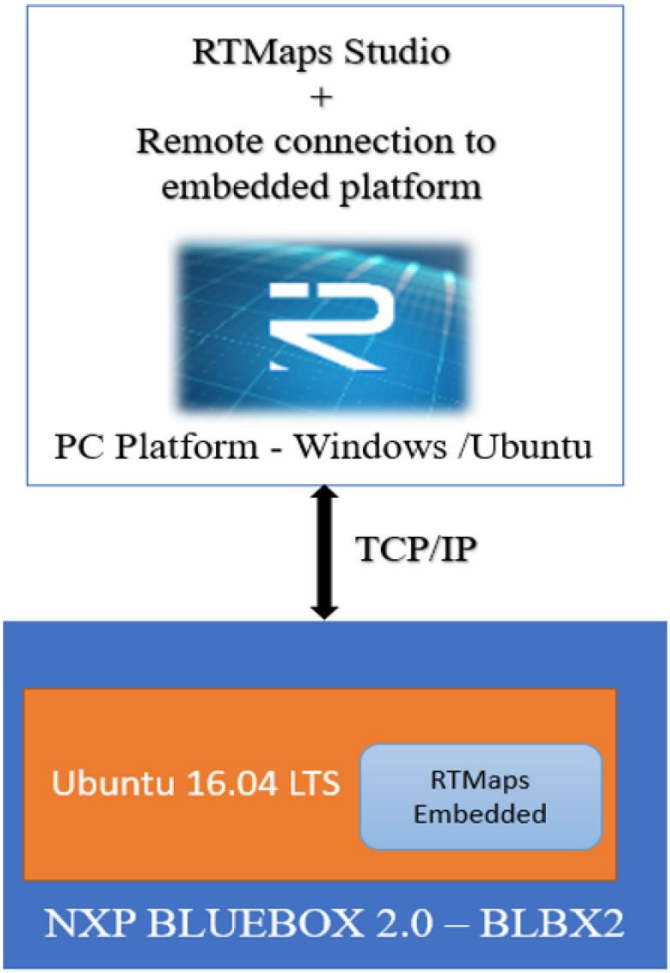


Fig. 4.7. RTMap Setup with BLBX2.

RTMaps Runtime Engine is an easily deployable, multi threaded, highly optimized module designed in a context to be integrated with third-party applications and is accountable for all base services such as component registration, buffer management, time stamping threading, and priorities.

RTMaps Component Library consists of the software module which is easily interfaceable with the automotive and other related sensors and packages such as Python, C++, Simulink models, and 3-d viewers, etc, responsible for the development of an application.

RTMaps Studio is the graphical modeling environment with the functionality of programming using Python packages. The development interface is available for the windows and ubuntu based platforms. Applications are developed by using the modules and packages available from the RTMaps Component library.

RTMaps Embedded is a framework which comprises of the component library and the runtime engine with the capability of running on an embedded x86 or ARM capable platform such as NXP Bluebox, Raspberry Pi, DSpace MicroAutobox, etc. RTMaps embedded v4.5.3 platform is tested with NXP Bluebox, it is used independently on the Bluebox, and with the RTMaps remote studio operating on a computer thus providing the graphical interface for the development and testing purpose. The connection between the Computer running RTMaps Remote studio and the embedded platform can be accessed via a static TCP/IP as shown in Fig. 4.7. which is RTMap setup with Bluebox 2.0.

4.5 Different Deep Learning Frameworks

4.5.1 TensorFlow

It is a free and open-source software library for differentiable programming and data flow a range of different tasks. TF [58] is used for machine learning applications such as neural networks, CNNs or DNNs. It is mostly used for research and production both, at Google. It provides a variety of different toolkits that allow you to construct CNN/ DNN models. We can use lower-level APIs to build models by defining a series of mathematical operations and higher-level APIs such as tf.estimator, to specify predefined architectures, such as linear regressors or NN/CNNs. It comes with two highly useful tools that are:

Fig. 4.8. Different DL Frameworks

Tensor Board: is mainly used for data visualization of network architecture and performance.

TensorFlow: It is used for rapid prototyping of models, and it has various standard API that are used to design modular networks.

4.5.2 Keras

It is another widely used deep learning framework. It [62] is primarily used for classification, speech recognition, and text generation and summarization. It is a high-level neural networks API, written in Python and capable of running on top of TensorFlow, CNTK, or Theano. It is used if deep learning library allows for easy and

fast prototyping (through user friendliness, modularity, and extensibility). It supports both CNN and RCNN, as well as combinations of the two and it runs seamlessly on CPU and GPU.

4.5.3 Caffe

It [48] is a deep learning framework made with expression, speed, and modularity in mind. It is developed by Berkeley AI Research (BAIR) and by community contributors. It is written in C++ and has Python and Matlab bindings. It is popular in research industries for its quick prototype deployment. It can process over 60 million images per day with a single NVIDIA K40GPU. It supports C, C++, Python, and MATLAB and command line interface.

4.5.4 PyTorch

Implementing or developing a CNN/DNN models is quite hard. requires a lot of skill, knowledge and kind of hard to implement. With a python at the back end running, Python APIs [59], and a good community support it is easy to implement ML/DL models. In contrast to TF, Caffe and some other frameworks they are not able to support the python CNN/DNN models and would not support python libraries such as numpy, scipy, scikit-learn, cython, etc. PyTorch [58, 60] deep learning library has one of purported benefits is that it is a DL library that is python based and supported mainly with python. It allows you to do the following:

- Dynamic data structures inside the network where we can have number of inputs at any given point during training in PyTorch.
- Networks here are modular. Each part is implemented, and debugged separately, unlike a TF monolithic construction.
- It also has features such as dynamic computational graph construction in contrast to the static computational graphs present in TF and Keras.

It allows us to write a lot of codes very quickly and easily without great losses in performance during training. In this thesis, all the DNN architectures are developed under the python and pytorch framework on the top of it.

4.6 Pytorch Packages Used

4.6.1 Autoaugment

In AutoAugment [74]: Learning Augmentation Policies from Data research, we explore a reinforcement learning algorithm which increases both the amount and diversity of data in training dataset which is already there. Data augmentation is used to teach a model about image invariances in the data or invent more artificial data. In this domain, it makes a CNN invariant to the important symmetries, improving a CNN performance. Instead of using a hand-designed data augmentation policies, we use autoaugment, reinforcement learning policy method, to find the optimal image transformation policies from the data itself.

AutoAugment algorithm achieved a good performance on many different competitive datasets such as CIFAR-10, CIFAR-100, Imagenet, etc, for instance it achieved 98.52% on CIFAR-10 dataset. The policies with the best performance results are included in the paper [74]. In this research this method was also implemented but it was left as we were not sure if this facing a underfitting problem with our proposed architectures implementation with CIFAR-10 but it is expected to perform better with a large dataset such as ImageNet.

4.6.2 Livelossplot

Livelossplot is a python and a matplotlib based package which is a CNN/DNN model visualization tool. It is supported by Keras, PyTorch and other frameworks and an open source python package by Piotr Migda, and others. Visualization allows us to keep track of the training process of our DNN model. It is really easy to implement

and easy to decipher information from it with help of live graphs after each epoch, in comparison to tensor board. It is easy to setup and configure. This package is used in the thesis research through out for visualization of the results obtained. It provides with a live CNN training and validation losses and accuracy. It aid hugely, in rapid training of different variations of proposed architecture and baseline architectures. It outputs a model log loss and accuracy as shown in Fig. 8.2.

5. DESIGN SPACE EXPLORATION

This chapter discusses the various DSE methodologies [5-8, 28-33] and techniques [50, 51, 53, 55, 61, 75, 78] which laid the foundation in developing this whole thesis and proposed architectures that are High Performance SqueezeNext and Shallow SqueezeNext.

5.1 Methods to Improve DNN Performance

The awareness of the general methods [33, 52] for the performance improvement of a DNN is established before hand. The performance can be improved in the following ways.

1) Improve the performance with data. This refers to collect and/or invent more data, improve the quality of the data, data augmentation [35, 80], and feature selection techniques.

2) Improve the performance with the architecture tuning which can be done by model diagnostics, tuning or tweaking with the following techniques such as weight initialization, learning rate, activation functions, network topology, regularization, different optimization and loss techniques.

3) Improve the performance with the architecture modification. In this method, new architecture can be inspired by the literature review, benefits of the existing architectures and re-sampling techniques.

4) Improve the performance with ensembles which include the following possible ways that are to combine models, combine views, and stacking.

5.2 Architecture Tuning

The following ideas are used in this thesis for the DSE of DNNs and tuning the proposed architectures:

1) Different Learning Rate Schedule.

2) Save and Load checkpoint.

3) Use of different optimizers.

4) Use of different activation functions.

5.3 Different Learning Rate Schedule Methods

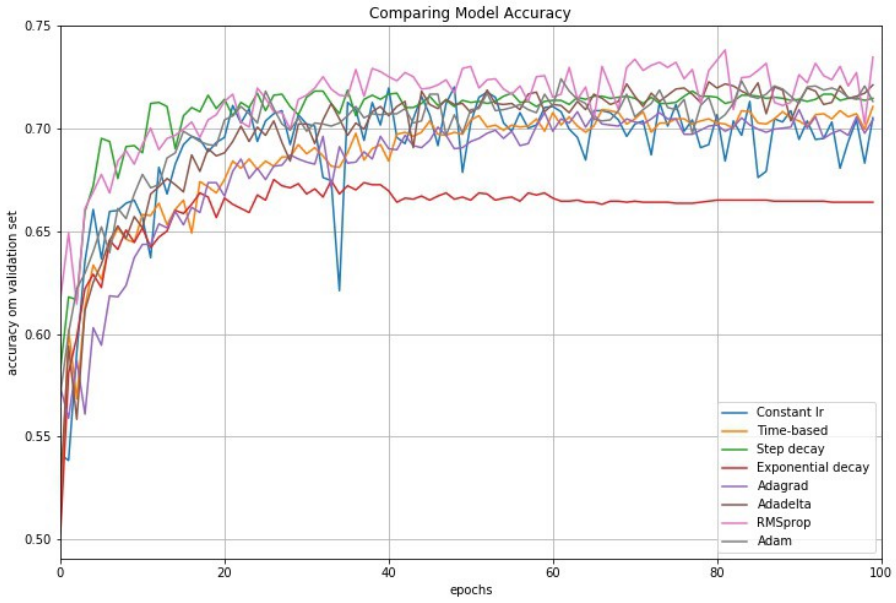


Fig. 5.1. Comparison of Different LR Scheduling Methods and Optimizers.

LR schedules [77] seek to adjust the LR during training by reducing the learning rate according to a pre-defined schedule. Common LR schedules include time-based decay, step decay, exponential decay, and cosine annealing. Fig. 5.1 illustrates step decay based learning rate performs better than other learning rate schedule methods.

5.4 Save & Load Checkpoint Method

For improving accuracy, save and load the model method was used. Optimizer state dictionary and sometimes, other items such as epochs, loss, accuracy, net module, embedding layers, etc are also saved and loaded. The optimizer state dictionary contains additional updated information of buffers and parameters. Layers with only the learnable parameters have entries in the model state dictionary while the optimizer state dictionary contains information about the optimizer's state and all the hyper parameters used. For the efficient model size and speed, the models state dictionary is only saved and loaded.

In Pytorch, torch.save() & torch.load() functions are used for saving and loading the state dictionaries in a checkpoint file. We can save checkpoint file in the format of .pkl, .ckpt or .pth, saving with .ckpt or .pth is recommended.

5.5 Use of Different Optimizers

In this thesis, different optimizers [21, 45, 52] were implemented on proposed architectures based on the insights. Refer [52] for the mathematical form or equations of the optimizers. All the different optimizers are compared in Fig. 5.2.

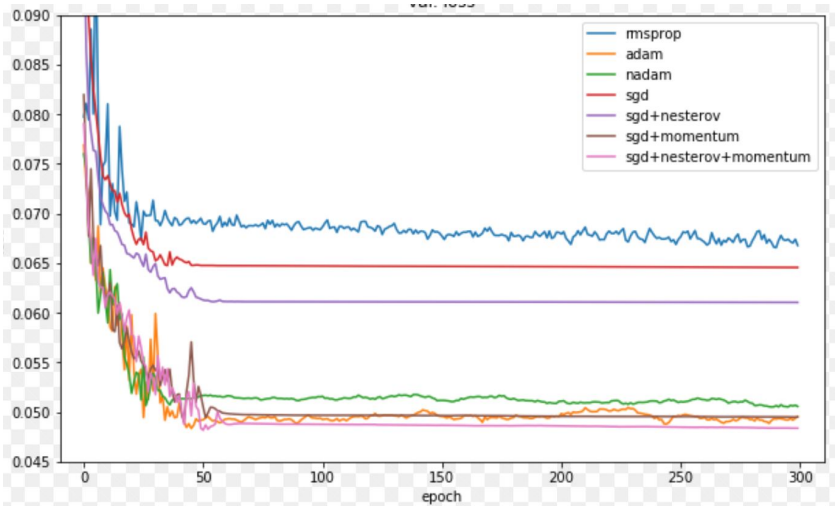


Fig. 5.2. Comparison of Different Optimizers.

5.5.1 SGD

SGD performs a parameter update that means there is one update at a time, for each of the training examples. But, the problem with SGD is the high variance with frequent updates and this causes the SGD optimizer to fluctuate heavily, further complicating the model convergence to the exact minima. This problem is usually referred to as navigating through ravines. We introduce a term called momentum to solve the ravines problem. Another term which is called nestrov is also used to help accelerate SGD in the relevant direction. As a result of introduction of these new terms our model converge faster. We use this variation of SGD which is includes to SGD optimizer with momentum, decay and nestrov terms.

5.5.2 Adagrad

Adagrad is responsible to make major and minor updates for infrequent and frequent parameters, respectively. So this optimizer is well-suited for dealing with sparse data. It updates the LR and uses every time a different LR for every parameter based on the values of past gradients. Therefore, manually tuning of LR is not required for this optimizer which is the key advantage of Adagrad. But, the weakness of it is that the LR kept on always decaying and decreasing because of the accumulation of squared gradients of each term. Further, in the later stages of DNN training it causes LR to shrink. This shrinkage eventually stops learning entirely when training further, on more number of epochs. This leads to extremely slow convergence and more over fitting.

5.5.3 Adadelta

Adadelta aims to reduce the aggressive and monotonically, decreasing LR. It restricts the accumulated past gradients to some fixed size instead of accumulating all the past squared gradients. Here, the sum of gradients is defined as a decaying

average of all past squared gradients and the running averages. The running average in Adadelta depends on both, previous average and the current gradient. The major benefit of it that we do not need to set a default value of LR.

5.5.4 RMSprop

RMSprop refers to Root Mean Square propagation. It will keep moving average of the squared gradient for each term of the weight. It divides the LR by an exponentially decaying average of squared gradients or by square root of mean square of (w,t) which will make the DNN learn better. It suggests the momentum and the LR to be set at the default the values of 0.9 and 0.001, respectively. These parameter value and setup is used to train and test the proposed architectures.

5.5.5 Adam

Adam [70] has benefits of both Adagrad and RMSprop. It adapts to the LR not only based on the average first moment as in RMSProp optimizer but also it stores the second moments of gradients as used in Adagrad. In other words, it accumulates both exponentially decaying average of past squared gradients and exponentially decaying average of past gradients. Adam will compute adaptive learning rate for each parameter eliminating the need for manual declaration of default LR. It is favourably used by DNN community as the DNN model learning is fast and efficient. With these benefits, it also solves the problems of vanishing gradient, slow convergence and high variance. But while, training the proposed models during this research, it was observed that, initially, the algorithm performs poorly even with very low LR and it faces the problem of slow convergence.

5.5.6 Adamax

Adamax is a variant of Adam where its second order moment is replaced by infinite order moment. This infinite order norm according to the adam research paper [70] makes this optimizer more stable. Good default adamax optimizer values for LR, beta 1 and beta 2 are 0.002, 0.9, and 0.999, respectively.

5.5.7 Rprop

Rprop is similar to backprop with the advantages such as fast training, would not require to specify any free parameter values, and adapts the step size for each weight ,dynamically. It tries to solve the problem of widely varied magnitudes of the gradients. The disadvantage of Rprop is that it is not able to perform well on sparse datasets, mini-batches weight updates and large datasets. more complex algorithm to implement. It generally requires large batch updates and the step sizes jump around too much and updates work badly if there is too much randomness.

5.5.8 Adabound

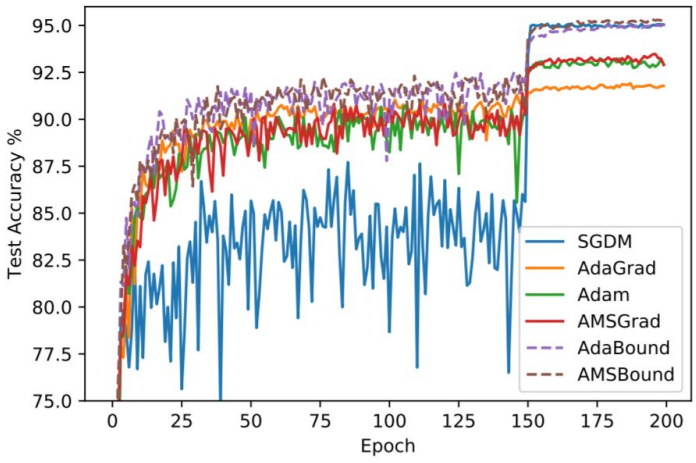


Fig. 5.3. Comparison of Adabound with Other Optimizers.

Adabound [45] is a recently introduced new optimizer algorithm. According to the research paper, it employs dynamic bounds on their LRs. Due to this it achieve a smooth transition to SGD from a Adam like optimization in the beginning. This maintains the advantageous properties of adaptive methods such as rapid initial progress and hyper parameter insensitivity. The lower bound and upper bound of adabound are always changing so that it is kind of transforming from initially Adam optimizer to SGD optimizer during the DNN training. This is suggested as the second best choice for a optimizer by this research study as it shows better results in comparison to other counter parts unless the model size is not a strict constraint.

5.6 Different Activation Functions

All the different activation functions implemented in the thesis study is discussed in the following sections. For more detailed reference or information about these activations functions refer this paper [90].

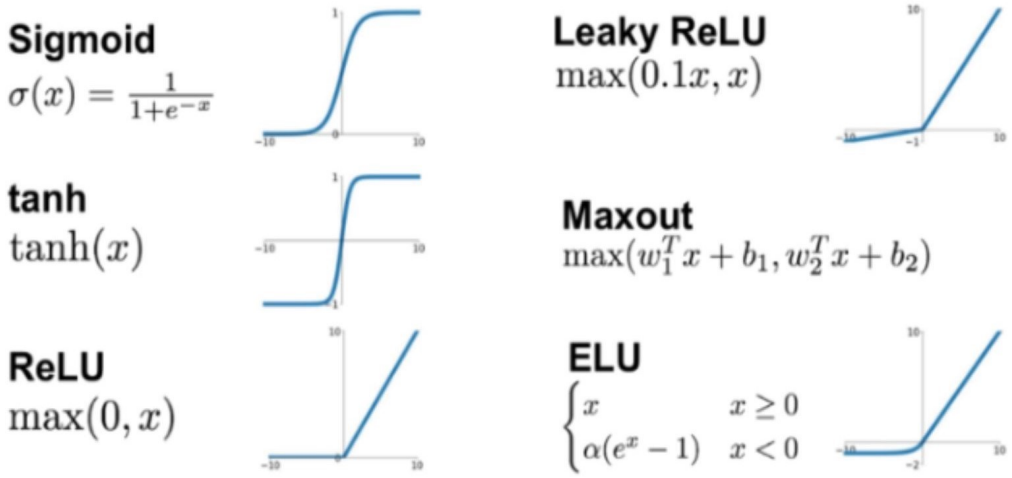


Fig. 5.4. Different Activation Functions for a CNN.

5.6.1 Sigmoid

Sigmoid function range is like a 'S' shaped curve activation function which lies between a range of 0 and 1. It is easy to implement but it has major problems such as vanishing gradient problem and, not zero centered output which made this activation loss its place in the DNN community. It makes too frequent and big gradient updates in different directions, resulting in harder optimization of DNN. Further it will saturate, kill gradients and will have slow convergence problems.

5.6.2 Tanh

Tanh is a better activation than sigmoid and analogous to sigmoid logistically. It solves one of the problem of sigmoid of zero centered output. The range of it is in between -1 to 1. It is also a 'S' shaped curve function. Additionally, it is differentiable, monotonic but here derivative is not monotonic. It is always preferred over sigmoid. It is usually used in feed forward nets. But, it suffers from vanishing gradient problem.

5.6.3 Rectified Linear Units (ReLU)

ReLU is most used activation function these days. It is not a linear as it sounds. It provides the similar benefits as of sigmoid but it is better than sigmoid and tanh. The mathematical formula is max (0,z). It involves simpler mathematical operations rectifies the vanishing gradient problem, and also, less computationally expensive. It has both function and derivative monotonic. But, it still has problem that it can blow up the activation function due to its wide range [0, inf). The negative values become zero resulting in a decreased ability of DNN to fit and also, it would not map the negative parts properly.

5.6.4 LeakyReLU

LeakyReLU is used to solve the problem with ReLU of dying gradients. It increases the range of the ReLU function from (-inf, inf). It contains the advantages of the ReLU. It causes a weight update which will not activate it on any data point again. It introduces a small slope to keep the gradient updates alive to avoid the dying gradient problem.

5.6.5 Maxout

Maxout is another variant made form of both ReLU and Leaky ReLU. It is a layer where the activation function is the max of the inputs. In it, inputs are not dropped to the maxout layer. It unit can learn a piecewise linear and convex function with up to k pieces. The maxout activation facilitates the training of dropout with large gradient updates and it simply is the max value of the function in mathematical terms.

5.6.6 Exponential Linear Unit (ELU)

ELU [46, 47] is an activation function that tends to converge to zero faster and produce more accurate results. It has an extra alpha constant, a positive number. It is very similar to RELU except for negative inputs. It slowly, becomes smooth as its output equal to -α in contrast to ReLU whole function sharply smooths. But, it can blow up the activation function too, for x greater than zero with the a range of [0, inf]. It converges to good model accuracy but it blows up in the later stages of DNN training.

6. HIGH PERFORMANCE SQUEEZENEXT

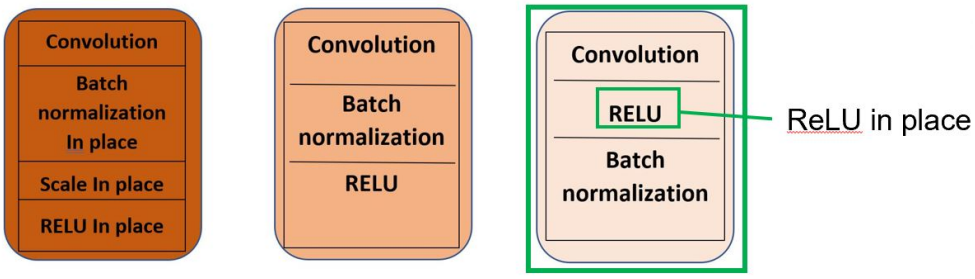


Fig. 6.1. Extreme Right Illustrates the Proposed High Performance SqueezeNext Basic-block Module.

The proposed High Performance SqueezeNext architecture [88, 90] is inspired [7, 8, 31, 32, 36, 37] from the baseline SqueezeNext [6] and the inspiration for implementation of the basic block module and ELU implementation within the proposed architecture is taken from the other two other papers [46, 47].

This architecture uses a different structure of basic-block module shown in Fig. 6.1 then, the baseline SqueezeNext and SqueezeNext pytorch implementation of basic-block structure. The proposed high performance architecture basic block is compatible with pytorch. The bottleneck module is chosen over fire module [29, 30, 33] for proposed high performance squeezenext architecture because as explained in the related architecture section in the Chapter 3 because of the four stage implementation of batch normalization [32] layers with in place operations.

The bottleneck module has a better parameter reduction than SqueezeNet and Resnet module due to the fact that it uses a two stage bottleneck module to reduce the number of input channels down to 3x3 convolution. Further, in SqueezeNext 3X3 convolution in comparison to other modules is decomposed into 3x1 and 1x3 convolutions (orange blocks) which in turn, again reduces the number of parameters,

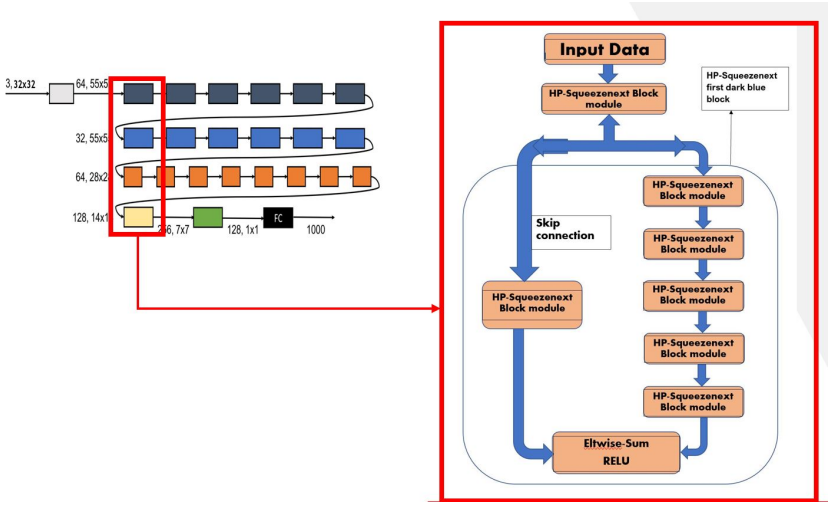


Fig. 6.2. Four Stage Implementation (6,6,8,1) with Basic Building Block Structure 1 for Proposed High Performance SqueezeNext Architecture.

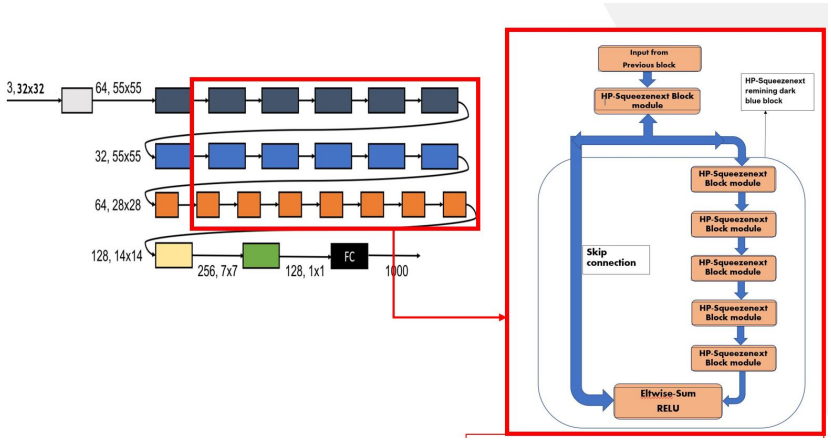


Fig. 6.3. Four Stage Implementation (6,6,8,1) with Basic Building Block Structure 2 for Proposed High Performance SqueezeNext Architecture.

followed by a 1x1 expansion module. The high performance squeezenext architecture comprises of bottleneck modules, activation layer, batch normalization layers, pooling layers (ceiling function is used instead of floor function) and a FC layer in the last with (1,1,1,1) four stage implementation configuration with 0.5x network width multiplier are used for better model performance.

The proposed architecture basic building block structures for the CIFAR-10, CIFAR-100 datasets [39] is shown in Fig. 6.2, and these structures are used in similar manner as used in the baseline architecture configuration as shown in Fig. 3.19 in Chapter 3. But the number of blocks in the four stage implementation configuration and network width multiplier are changed depending on the architecture end application. The basic-block module in Fig. 6.1 and the basic building block structure modules shown in Fig. 6.2, together implemented in (1,1,1,1) four stage implementation configuration with network width multiplier 0.5x forms the complete proposed high performance SqueezeNext architecture.

Table 6.1.
High Performance SqueezeNext Architecture with (1,2,8,1) Four Stage Configuration.

Layer Name	**Input Size** Wi X Hi X Ci	**Padding** Pw X Ph	**Stride**	**Filter size** Kw X Kh	**Output size** W0 X H0 X C0	**Repeat**	**Parameters**
Convolution 1	32x32x3	0x0	1	3x3	30x30x64	1	1792
Convolution 2	30x30x64	0x0	1	1x1	30x30x16	1	1040
Convolution 3	30x30x16	0x0	1	1x1	30x30x8	1	136
Convolution 4	30x30x8	0x1	1	1x3	30x30x16	1	400
Convolution 5	30x30x16	1x0	1	3x1	30x30x16	1	784
Convolution 6	30x30x16	0x0	1	1x1	30x30x32	1	544
Convolution 32	30x30x32	0x0	2	1x1	30x30x32	1	1056
Convolution 33	15x15x32	0x0	1	1x1	15x15x16	1	528
Convolution 34	15x15x16	0x1	1	1x3	15x15x32	1	1568
Convolution 35	15x15x32	1x0	1	3x1	15x15x32	1	3104
Convolution 36	15x15x32	0x0	1	1x1	15x15x64	1	2112
Convolution 37	15x15x64	0x0	1	1x1	15x15x32	1	2080
Convolution 38	15x15x32	0x0	1	1x1	15x15x16	1	528
Convolution 39	15x15x16	1x0	1	3x1	15x15x32	1	1568
Convolution 40	15x15x32	0x1	1	1x3	15x15x32	1	3104
Convolution 41	15x15x32	0x0	1	1x1	15x15x64	1	2112
Convolution 62	15x15x64	0x0	2	1x1	15x15x64	1	4160
Convolution 63	8x8x64	0x0	1	1x1	8x8x32	1	2080
Convolution 64	8x8x32	1x0	1	3x1	8x8x64	1	6208
Convolution 65	8x8x64	0x1	1	1x3	8x8x64	1	12352
Convolution 66	8x8x64	0x0	1	1x1	8x8x128	1	8320
Convolution 67	8x8x128	0x0	1	1x1	8x8x64	7	57792
Convolution 68	8x8x64	0x0	1	1x1	8x8x32	7	14560
Convolution 69	8x8x32	1x0	1	3x1	8x8x64	7	43456
Convolution 70	8x8x64	0x1	1	1x3	8x8x64	7	86464
Convolution 71	8x8x64	0x0	1	1x1	8x8x128	7	58240
Convolution 102	8x8x128	0x0	2	1x1	8x8x128	1	16512
Convolution 103	4x4x128	0x0	1	1x1	4x4x64	1	8256
Convolution 104	4x4x64	0x1	1	1x3	4x4x128	1	24704
Convolution 105	4x4x128	1x0	1	3x1	4x4x128	1	49280
Convolution 106	4x4x256	0x0	1	1x1	4x4x256	1	65792
Convolution 107 (Spatial Resolution)	4x4x256	0x0	1	1x1	4x4x128	1	32896
Average Pool	4x4x256	-	-	-	4x4x256	1	-
Fully Connected Convolution	1x1x128	0x0	1	1x1	1x1x10	1	1290

* Wi, Hi, Ci refer to input width, height and number of channels, Pw, Ph refer to padding width and height, Kw, Kh refer to filter or kernel width and height, W0, H0, C0 refer to output width, output height and output number of channels

7. SHALLOW SQUEEZENEXT

The proposed Shallow SqueezeNext architecture [89, 91] is a compact DNN architecture. It is inspired from SqueezeNext, SqueezeNet and Mobilenet architectures [5-8]. It is based on the SqueezeNext architecture and a shallower architecture.

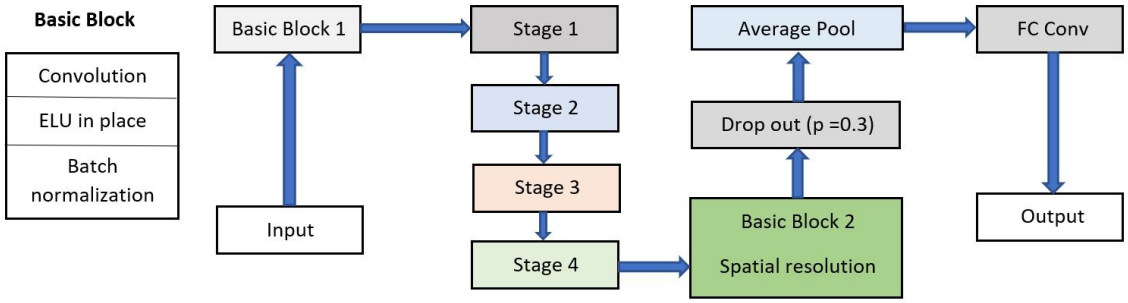


Fig. 7.1. Illustration of Basic-Block & Proposed Shallow SqueezeNext architecture.

It comprises of bottleneck modules [6] which are further made up of basic blocks arranged in a four stage configuration followed by a spatial resolution layer, average pooling layer and a FC layer. The architecture implements SGD optimizer (results obtained on it are better than other optimizers although other optimizers are too implemented). Further, the bottleneck module, shown on the right side of Fig. 3.17, comprises of a 1x1 convolution, second 1x1 convolution, 3x1 convolution, 1x3 convolution and then, a 1x1 convolution.

The basic block consists of a convolution layer, Relu in place, and batch normalization layer, as shown in the left side of Fig. 7.1. These basic blocks form bottleneck modules which are together arranged in the four stage implementation configuration along with a dropout layer are shown in Fig. 7.3. It is concluded with the descriptions of the two model shrinking hyper parameters such as the width

multiplier and resolution multiplier in the following subsections. The right side of Fig. 7.1 illustrates the proposed architecture with (1,2,8,1) four stage configuration. Fig 7.2 illustrates the Shallow SqueezeNext bottleneck module comprising of Shallow SqueezeNext basic blocks.

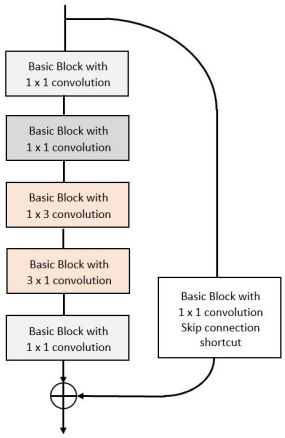


Fig. 7.2. Illustration of Shallow SqueezeNext's Bottleneck Module.

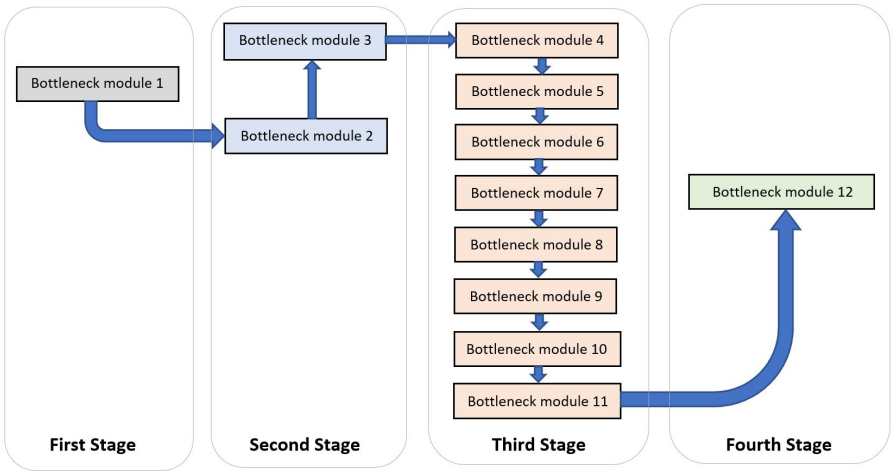


Fig. 7.3. Illustration of Four stage (1,2,8,1) Configuration of Shallow SqueezeNext Architecture.

7.0.1 Dropout Layer

Dropout [34] is a technique used to improve over fit on NNs. It is a regularization method that approximates training a large number of NNs with different parallel architectures. Large NNs trained on relatively small datasets can over fit the training data. The approach to reduce over fitting is to fit all possible different NNs on the same dataset and to average the predictions from each model.

Normally some DL models use dropout on the FC layers, but it is also possible to use dropout after the max-pooling layers. In the proposed architecture, the dropout layer is used before the spatial resolution layer followed by the average pooling layer.

7.0.2 Resolution Multiplier

This hyper-parameter [7] is used to reduce the computational cost of a NN. We apply this to the input image and the internal representation of every layer is subsequently reduced by the same multiplier. In practice we implicitly set value of the input resolution. The value with $<$ than 1 generate reduced computational DNN models/ architectures. It reduces the computational cost by the square of the resolution multiplier. Values used in this thesis are 0.125, 0.5, 0.75, 1.0, 1.5 & 2.0.

7.0.3 Width Multiplier

Width multiplier [7] is used in order to construct these smaller and less computationally expensive models. The role of the width multiplier is to thin a network uniformly at each layer. The typical settings of width multiplier are 1, 0.75, 0.5 and 0.25. Width multiplier has the effect of reducing computational cost and the number of parameters quadratically by roughly twice the power of the width multiplier term. Width multiplier can be applied to any model structure to define a new smaller model with a reasonable accuracy and size trade off.

Table 7.1.
SSqNxt Architecture with (1,2,8,1) Four Stage Configuration.

Layer Name	**Input Size** Wi X Hi X Ci	**Padding** Pw X Ph	**Stride**	**Filter size** Kw X Kh	**Output size** W0 X H0 X C0	**Repeat**	**Parameters**
Convolution 1	32x32x3	0x0	1	3x3	30x30x64	1	1792
Convolution 2	30x30x64	0x0	1	1x1	30x30x16	1	1040
Convolution 3	30x30x16	0x0	1	1x1	30x30x8	1	136
Convolution 4	30x30x8	0x1	1	1x3	30x30x16	1	400
Convolution 5	30x30x16	1x0	1	3x1	30x30x16	1	784
Convolution 6	30x30x16	0x0	1	1x1	30x30x32	1	544
Convolution 32	30x30x32	0x0	2	1x1	30x30x32	1	1056
Convolution 33	15x15x32	0x0	1	1x1	15x15x16	1	528
Convolution 34	15x15x16	0x1	1	1x3	15x15x32	1	1568
Convolution 35	15x15x32	1x0	1	3x1	15x15x32	1	3104
Convolution 36	15x15x32	0x0	1	1x1	15x15x64	1	2112
Convolution 37	15x15x64	0x0	1	1x1	15x15x32	1	2080
Convolution 38	15x15x32	0x0	1	1x1	15x15x16	1	528
Convolution 39	15x15x16	1x0	1	3x1	15x15x32	1	1568
Convolution 40	15x15x32	0x1	1	1x3	15x15x32	1	3104
Convolution 41	15x15x32	0x0	1	1x1	15x15x64	1	2112
Convolution 62	15x15x64	0x0	2	1x1	15x15x64	1	4160
Convolution 63	8x8x64	0x0	1	1x1	8x8x32	1	2080
Convolution 64	8x8x32	1x0	1	3x1	8x8x64	1	6208
Convolution 65	8x8x64	0x1	1	1x3	8x8x64	1	12352
Convolution 66	8x8x64	0x0	1	1x1	8x8x128	1	8320
Convolution 67	8x8x128	0x0	1	1x1	8x8x64	7	57792
Convolution 68	8x8x64	0x0	1	1x1	8x8x32	7	14560
Convolution 69	8x8x32	1x0	1	3x1	8x8x64	7	43456
Convolution 70	8x8x64	0x1	1	1x3	8x8x64	7	86464
Convolution 71	8x8x64	0x0	1	1x1	8x8x128	7	58240
Convolution 102	8x8x128	0x0	2	1x1	8x8x128	1	16512
Convolution 103	4x4x128	0x0	1	1x1	4x4x64	1	8256
Convolution 104	4x4x64	0x1	1	1x3	4x4x128	1	24704
Convolution 105	4x4x128	1x0	1	3x1	4x4x128	1	49280
Convolution 106	4x4x256	0x0	1	1x1	4x4x256	1	65792
Convolution 107 Spatial Resolution	4x4x256	0x0	1	1x1	4x4x128	1	32896
Dropout (p=0.5)	4x4x256	-	-	-	4x4x256	1	-
Average Pool	4x4x256	-	-	-	4x4x256	1	-
Fully Connected Convolution	1x1x128	0x0	1	1x1	1x1x10	1	1290

* Wi, Hi, Ci refer to input width, height and number of channels, Pw, Ph refer to padding width and height, Kw, Kh refer to filter or kernel width and height, W0, H0, C0 refer to output width, output height and output number of channels

8. RESULTS

8.1 Proposed High Performance SqueezeNext

The obtained results for all the DNN architectures including both of the proposed architectures are discussed in this section. The proposed architecture is trained and tested from scratch on the CIFAR-10, CIFAR-100 dataset to improve the overall performance of the proposed architecture. All the results obtained in this thesis were implemented with the following common hyperparameter values: 0.1, batch size: 128, weight decay: 5e-4, total number of epochs: 200, standard cross entropy loss function and with a live accuracy and loss graphs with the help of livelossplot package.

Table 8.1.
Proposed HPSqnxt Model Performance Improvement

Model	**Accuracy%**	**Model Size(MB)**	**Model speed (sec)**
SqueezeNet-v1-0(baseline)	**78.1**	2.75	7
SqueezeNext-23-1x-v1(baseline)	**87.56**	2.59	23
Proposed modified implementation of SqueezeNext-23-1x-v1	**92.25**	**5.14**	48
Proposed HP-SqueezeNext-21-1x-v2	**92.05**	**2.60**	18
Proposed HP-SqueezeNext-06-0.50x-v1	82.44	0.370	7
Proposed HP-SqueezeNext-06-1x-v1	86.82	1.24	8
Proposed HP-SqueezeNext-06-0.75x-v1	82.86	1.24	8

*All results are 3 average runs with SGD, LR is 0.1

8.1.1 Proposed HPSqnxt Model Accuracy Improvement

Other existing algorithms and methodologies have attained better accuracy than the SqueezeNext architecture's modified version that is 92.09% accuracy, shown in Table 8.1. However, all those ML algorithms use transfer learning techniques, in which the model is first trained on a large dataset, ImageNet and then pre-trained model is fine-tuned on a smaller datasets like CIFAR-10, CIFAR-100. Additionally these architectures also use data augmentation techniques. The transfer learning technique provides better accuracy than a network trained from scratch due to that reason.

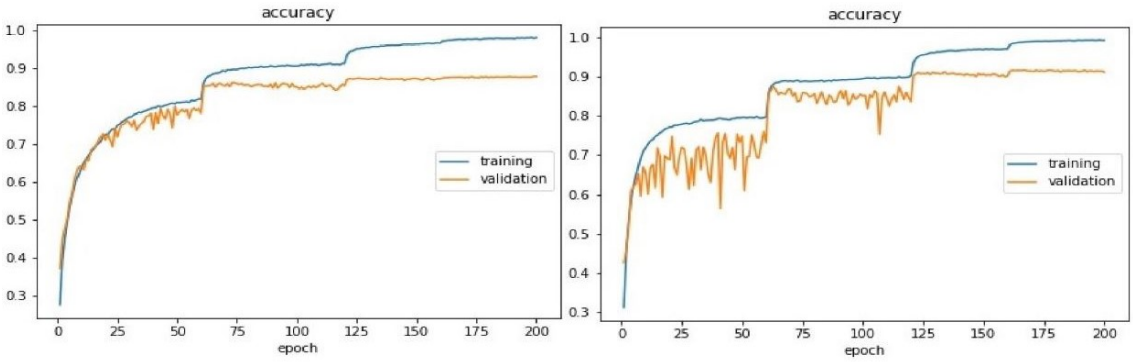


Fig. 8.1. SqNxt Baseline Accuracy, SqNxt Pytorch's (Iuuuyi) Accuracy.

However, training the network using the small datasets takes less time and computation power. Therefore, the models mentioned in this thesis, are the networks which were trained from scratch on the CIFAR-10, CIFAR-100 datasets and without use of any data augmentation. To improve the accuracy of the squeezenext pytorch architecture, the save and load checkpoint method is implemented along with the architecture modification with a kernel size 3x3 and stride 1. Also, the specific step time LR schedule with the exponential update is also used. Table 8.1 and Fig. 8.1, Fig. 8.2 compares the results obtained for the modified architecture, the proposed high performance SqueezeNext architecture with baseline SqueezeNet and SqueezeNext architectures.

8.1.2 Model Size & Speed Improvement

The model speed in this thesis refers to the per epoch time cost of training and testing the architecture on CIFAR-10 & CIFAR-100 datasets deployed on GPUs (GTX 1080 & RTX 2080 Ti). In general, more powerful hardware (better GPU or multiple GPUs), architecture pruning, and methods can be used to improve a CNN model size and speed. The CIFAR-10, CIFAR-100 datasets are quite small as compared to Imagenet so the model depth, as well as the width, is reduced for better model performance.

The proposed high performance SqueezeNext is implemented with low network depth and width multipliers, in-place activation layers, element-wise operations, no max-pooling layers were used only average pooling is used in the last just before the FCC layer. All the HP-SqueezeNext architectures uses (1,1,1,1) four stage implementation configuration for a small model size deployment with a different network width. Along with the following hyper parameters such as SGD optimizer with momentum and nestrov values equal to 0.9 and true, Step LR decay schedule comprising of four different LRs with an exponential LR update were used to train and test the CIFAR-10, CIFAR-100 datasets.

Table 8.2.
Proposed High Performance SqueezeNext CIFAR100 Results

Model	Width, Resolution	Accuracy%	Model Size(MB)	Model speed (sec)
SqueezeNext (Baseline)	1.0x, 6681	**60.37**	**5.2**	19
SqueezeNet (Baseline)	-, -	**51.27**	**6.4**	4
Proposed HPSqueezeNext-14-1.0x-v1	1.0x, 1281	**68.4**	**5.0**	14
Proposed HPSqueezeNext-25-1.0x-v1	1.0x, 24161	**70.1**	**7.8**	25
Proposed HPSqueezeNext-23-1.0x-v1	1.0x, 22161	68.2	7.7	25
Proposed HPSqueezeNext-23-0.2x-v1	0.2x, 22161	51.5	0.803	25
Proposed HPSqueezeNext-14-1.5x-v1	1.5x, 1281	69.7	10.8	18
Proposed HPSqueezeNext-06-0.575x-v1	0.575x, 1111	60.678	1	7
Proposed HPSqueezeNext-06-0.4x-v1	0.4x, 1111	55.89	1	6
Proposed HPSqueezeNext-09-0.5x-v1	0.5x, 1141	62.72	1.1	8

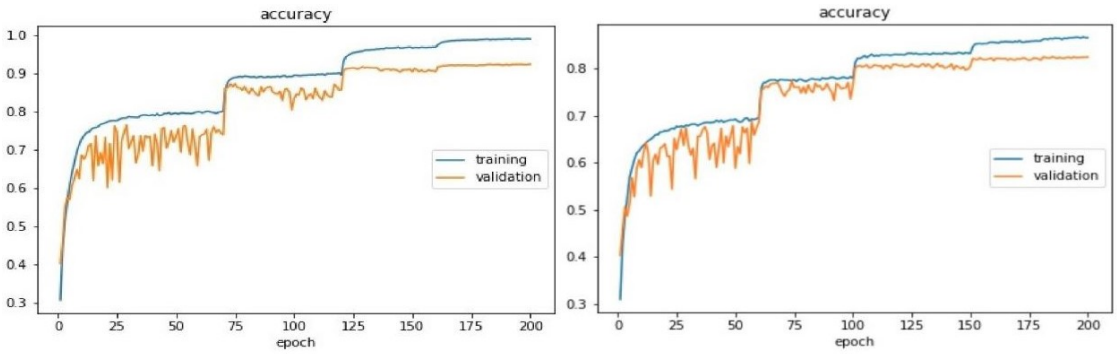


Fig. 8.2. SqueezeNext Pytorch's Modified Architecture Accuracy (Best Accuracy), High Performance SqueezeNext-06-1x-v1 Accuracy (Best Model Size & Speed).

Also, optimizer and other additional state dictionary were not saved in the model checkpoint file, only net state dictionary is saved. Minimum achieved model size is 370KB with a model speed of 7 seconds which is the average time cost for one epoch. The proposed architecture is trained and tested on CIFAR-10 dataset, initially, in 28 minutes, then to the squeezenext baseline architecture and squeezenext pytorch implementation which takes up to 1 hours 21 minutes and 2 hours 42 minutes, respectively. The proposed high performance architecture is able to achieve both better model size and model speed with decent accuracy. Though, a better model size and speed of 117 KB and 5 seconds respectively, is also achieved with a model width of 0.125x for the proposed high performance squeezenext architecture, but the model accuracy reduced to 61.87%. Fig. 8.2 shows the proposed architecture result with best model size and model speed. CIFAR-100 takes 10 -20 minutes more than the CIFAR-10 dataset on an average for training the model.

8.2 Proposed Shallow SqueezeNext Results

It can be observed that leveraging architectural modifications led to the generation of even more efficient network architectures, as evident by the Shallow

SqueezeNext having model sizes range from 4.2MB to just 0.115MB shown in Table 8.4. A better reduced model size is achieved from baseline SqueezeNext model size, 9.525MB to the reduced model size of the proposed Shallow SqueezeNext architecture, 0.115MB. Few major factors responsible for this model size reduction are different resolution and width multipliers. The ability to not only achieve very small model sizes but also fast runtime speeds has great benefits when used in resource-starved environments with limited computational, memory, and energy requirements.

The width and resolution multipliers are useful modifications for producing very small deep neural network architectures that are well-suited for embedded device scenarios without the need for compression or quantization. Therefore, Shallow SqueezeNext-06-0.125 is 22X smaller than SqueezeNext-23-1x-v1 (or 26X smaller than SqueezeNext v1.0). Implementation of in place operations such as ELU in place, Relu in place and elimination of the extra max pooling layers along with a suitable resolution and width multiplier makes this architecture more compact, efficient and flexible. With a change in resolution and width multiplier this architecture can be deployed with better accuracy with a trade off with the large memory size. Each model is trained from scratch on CIFAR-10, CIFAR-100 without the use of transfer learning method.

The network parameters for each model is saved and loaded from a checkpoint file for training and testing using pytorch save and load method for saving and loading a NN checkpoint model file. Most important factor of this architecture is that it can be readily implemented on real time systems with memory constraints. The dropout layer further, improved the accuracy of the architecture and further, Table 8.5 justifies why the specific dropout probability is chosen for the proposed Shallow SqueezeNext architecture. The naming format for Shallow SqueezeNext models in all the tables illustrates the Shallow SqueezeNext architecture with resolution multiplier followed by a width multiplier and the version number.

From the results of Table 8.2, it is observed that bottleneck module makes a

large difference along with appropriate use of multipliers makes the Shallow SqueezeNext with dropout layer more efficient. Table 8.5 illustrates results attained for different values of dropout layer probabilities implemented with the proposed Shallow SqueezeNext architecture. It shows the efficient results attained for the experiments and justifies the reason to choose the dropout probability p with value equal to 0.3.

8.2.1 Results Comparison with SqueezeNet & SqueezeNext Trained from Scratch on CIFAR-10

Table 8.3.
Proposed Shallow SqueezeNext Results Comparison with SqueezeNet & SqueezeNext Trained from Scratch on CIFAR-10.

Model	**Accuracy%**	**Model Size(MB)**	**Model speed (sec)**
SqueezeNet-v1.0 (Baseline)	79.59	3.013	4
SqueezeNet-v1.1 (Baseline)	77.55	**2.961**	4
SqueezeNext-23-1x-v1 (Baseline)	87.15	**2.586**	19
SqueezeNext-23-1x-v5	87.96	2.586	19
SqueezeNext-23-2x-v1	90.48	9.525	22
SqueezeNext-23-2x-v5	90.48	9.525	28
Proposed Shallow SqueezeNext-v1	82.44	**0.370**	7
Proposed Shallow SqueezeNext-v1	82.86	1.24	8

*All results are 3 average runs with SGD, LR is 0.1

Table 8.4.
Proposed SSqnxt Results with Different Resolution Multipliers.

Model	Resolution	Accuracy%	Model Size(MB)	Model speed (sec)
SqueezeNet-v1.0 (Baseline)	n/a	79.59	3.013	4
SqueezeNet-v1.1 (Baseline)	n/a	77.55	**2.961**	4
SqueezeNext-23-1x-v1 (Baseline)	6681	87.15	**2.586**	19
Proposed Shallow SqueezeNext-06-2x-v1	1111	89.35	**4.21**	21
Proposed Shallow SqueezeNext-06-2x-v1	1111	89.35	4.21	21
Proposed Shallow SqueezeNext-08-1x-v1	1221	77.48	2.96	4
Proposed Shallow SqueezeNext-10-1x-v1	1331	87.63	2.5863	23
Proposed Shallow SqueezeNext-12-1x-v1	1441	87.63	2.5863	23
Proposed Shallow SqueezeNext-14-1x-v1	1551	82.44	**0.370**	7
Proposed Shallow SqueezeNext-16-1x-v1	1661	82.86	1.24	8

*All results are 3 average runs with SGD, LR is 0.1

Table 8.5.
Proposed SSqNxt Results with Different Width Multipliers.

Model	Width	Accuracy%	Model Size(MB)	Model speed (sec)
SqueezeNet-v1.1 (Baseline)	n/a	77.55	**2.961**	4
SqueezeNext-23-1x-v1 (Baseline)	1.0x	87.15	**2.586**	19
Proposed Shallow SqueezeNext-06-0.125x-v1	0.125x	66.40	0.115	7
Proposed Shallow SqueezeNext-06-0.2x-v1	0.2x	72.16	0.141	8
Proposed Shallow SqueezeNext-06-0.2x-v1	0.2x	82.47	0.296	13
Proposed Shallow SqueezeNext-06-0.3x-v1	0.3x	77.87	**0.196**	**8**
Proposed Shallow SqueezeNext-12-0.4x-v1	0.4x	87.27	0.485	13
Proposed Shallow SqueezeNext-14-0.5x-v1	0.5x	88.96	0.772	15
Proposed Shallow SqueezeNext-06-0.6x-v1	0.6x	84.63	0.48	10
Proposed Shallow SqueezeNext-07-0.7x-v1	0.7x	88.10	0.704	12
Proposed Shallow SqueezeNext-06-0.8x-v1	0.8x	87.71	0.774	12
Proposed Shallow SqueezeNext-06-0.9x-v1	0.9x	86.25	0.950	12
Proposed Shallow SqueezeNext-12-1.0x-v1	1.0x	**88.28**	**0.557**	**19**
Proposed Shallow SqueezeNext-06-1.5x-v1	1.5x	82.44	2.44	17
Proposed Shallow SqueezeNext-06-2.0x-v1	2.0x	89.35	4.20	21

*All results are 3 average runs with SGD, LR is 0.1

Table 8.6.
Proposed Shallow SqueezeNext Results with Different Dropout Layer Probabilities.

Model	dropout p	Accuracy%	Model Size(MB)	Model speed (sec)
Proposed Shallow SqueezeNext-06-0.1x-v1	0.1	80.82	0.273	4
Proposed Shallow SqueezeNext-06-0.2x-v1	0.2	81.44	0.273	4
Proposed Shallow SqueezeNext-06-0.3x-v1	**0.3**	**81.87**	**0.273**	**4**
Proposed Shallow SqueezeNext-06-0.4x-v1	0.4	81.86	0.273	4
Proposed Shallow SqueezeNext-06-0.5x-v1	0.5	81.70	0.273	4

Table 8.7.
Proposed Shallow SqueezeNext Results with CIFAR-100 and Different Optimizers.

Model	Optimizer	Accuracy%	Model Size(MB)	Model speed (sec)
SqueezeNext (Baseline)	1.0x, 6681	**60.37**	**5.2**	19
SqueezeNet (Baseline)	-, -	**51.27**	**6.4**	4
Proposed Shallow SqueezeNext-14-1.0x-v1	**Adabound**	**62.12**	**6.90**	20
Proposed Shallow SqueezeNext-09-0.5x-v1	Adam	58.27	1.4	11
Proposed Shallow SqueezeNext-09-0.5x-v1	Adamax	33.73	1.4	13
Proposed Shallow SqueezeNext-09-0.5x-v1	Adagrad	35.57	1.0	9
Proposed Shallow SqueezeNext-09-0.5x-v1	Adabound	51.84	1.4	12
Proposed Shallow SqueezeNext-09-0.5x-v1	Adadelta	44.40	1.4	12
Proposed Shallow SqueezeNext-09-0.5x-v1	ASGD	57.39	1.0	9
Proposed Shallow SqueezeNext-09-0.5x-v1	RMSprop	27.73	1.4	11
Proposed Shallow SqueezeNext-09-0.5x-v1	Rprop	16.84	1.4	25
Proposed Shallow SqueezeNext-06-0.5x-v1	**SGD**	**59.69**	**1.1**	8

Table 8.8.
Proposed Shallow SqueezeNext Best Results

Model	Width, Resolution	Accuracy%	Model Size(MB)	Model speed (sec)
SqueezeNet-v1.0 (Baseline)	-	79.59	3.013	4
SqueezeNet-v1.1 (Baseline)	-	77.55	**2.961**	**4**
SqueezeNext-23-1x-v1 (Baseline)	-	87.15	**2.586**	**19**
Proposed Shallow SqueezeNext-14-1.5x-v1	1.5x, 1281	91.41	8.72	22
Proposed Shallow SqueezeNext-21-0.2x-v1	0.2x, 22141	90.27	1.814	27
Proposed Shallow SqueezeNext-06-0.575x-v1	0.575x, 1111	81.80	0.449	6
Proposed Shallow SqueezeNext-06-0.4x-v1	0.4x, 1111	81.97	**0.2717**	**9**
Proposed Shallow SqueezeNext-09-0.5x-v1	0.5x, 1141	87.73	0.531	11

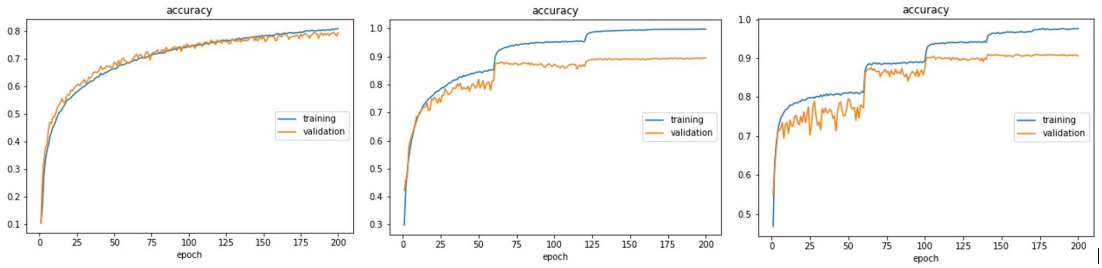


Fig. 8.3. SqueezeNet's Accuracy, SqueezeNext's Accuracy, Shallow SqueezeNext's Accuracy.

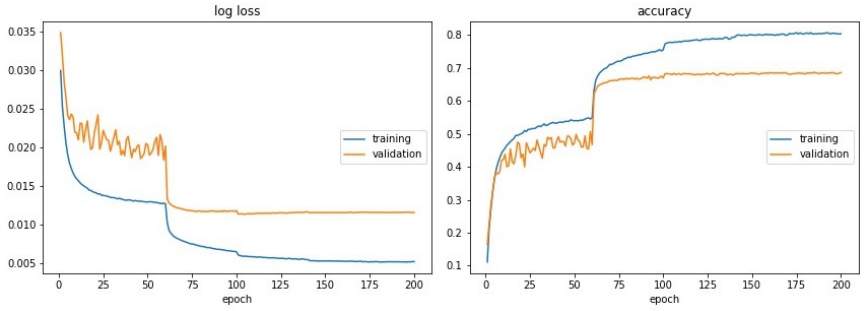


Fig. 8.4. Shallow SqueezeNext Accuracy with CIFAR-100

9. IMPLEMENTATION

9.1 BlueBox2.0 Implementation

The proposed architectures, HPSqnxt & SSqnxt are trained on GPU RTX 2080ti and then testing is done on bluebox2.0 using RTMaps studio. RTMaps provides support for Pytorch 3.7. Fig. 9.1 represents the process of deploying the proposed architecture's classifiers on bluebox2.0 development platform. RTMaps studio initiates a connection to the execution engine using TCP/IP which runs the software on BSP Linux OS installed on Bluebox2.0. RTMaps provides a python block to create and deploy python pytorch code. Python code for RTMaps comprises of three function definition birth(), core() and death().

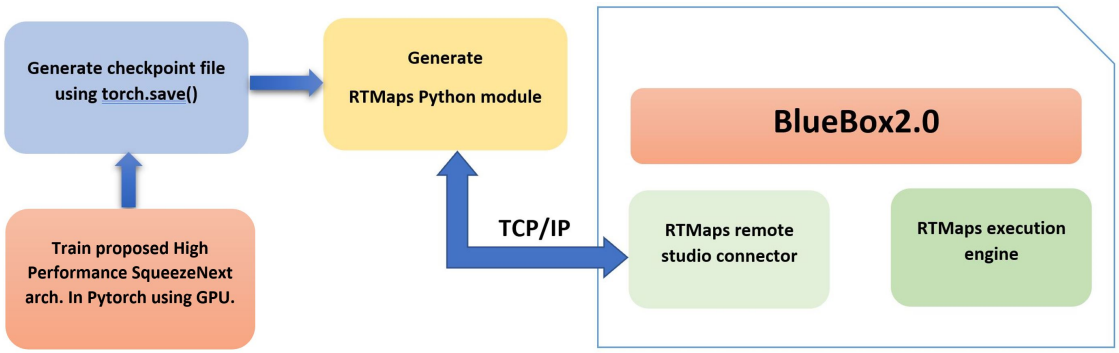


Fig. 9.1. Illustration of High Performance & Shallow SqueezeNext BlueBox2.0 Implementation.

Birth() module is executed one time for setting up and initializing the python environment. **Core()** is an infinite loop function to keep the code running continuously. **Death()** is used to perform cleanups and memory release after the python code terminates. Due to this ease and flexibility within RTmaps and organized structure makes it easy for developing a modular code. For maximum

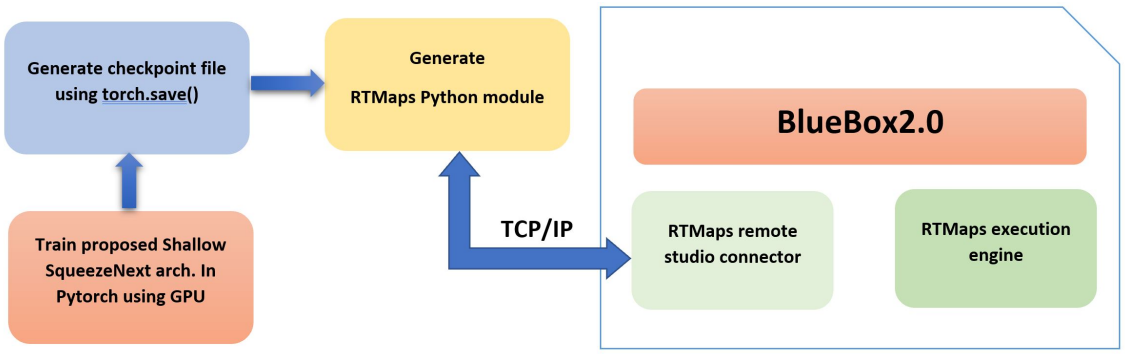


Fig. 9.2. Illustration of Shallow SqueezeNext BlueBox2.0 Implementation.

utilization of available 8 ARM Cortex-A72 cores to run the RTMaps embedded the LS2084A processor is used. The debug functionality and graphical interface the RTMaps embedded is connected with the RTMaps Studio in the desktop PC, illustrated in Fig. 9.2 using the remote engine connectivity provided over TCP/IP.

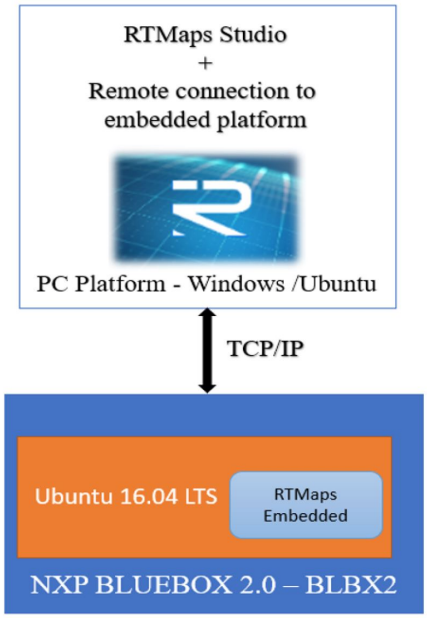


Fig. 9.3. Remote Engine Connectivity for BlueBox2.0 Implementation.

9.2 BlueBox2.0 Implementation Results

BlueBox2.0 [43] platform is where the image classifier is deployed and tested but the classifier is trained on GPU. The network parameters for all model of both architectures (High Performance SqueezeNext & Shallow SqueezeNext) is saved and loaded from a checkpoint file for training and testing using pytorch's save and load functions. Most important factor of this architecture is that it can be readily implemented on real time systems with memory constraints [31, 32, 36]. The dropout layer further, improved the accuracy of the architecture.

Table 9.1 compares the proposed architecture result with Squeezed CNN architecture [87] which shows the proposed architecture is a better and efficient architecture in terms of model accuracy and model size. All the architectures shown in Table 9.1 are trained and tested from scratch on CIFAR-10, CIFAR-100 datasets [39] without use of transfer learning method. Fig. 9.2 shows the graphical interfaces in RTMaps which comprises of python version 3.2 code block and an Image viewer block. Fig. 9.3 illustrates a prediction based on the image classifier of Squeezed CNN architecture [87].

Fig. 9.4 illustrates the HPSqnxt [88, 90] ground truth & predicted images & Fig. 9.5 illustrates the SSqnxt [89, 91] ground truth and predicted images outputs. Fig. 9.7 & Fig. 9.8 shows the testing results for HPSqnxt & SSqnxt deployed on BLBX2 by NXP. SSqnxt attained a model accuracy of 90.5%, shown in Table 9.1. HPSqnxt performs better than all other DNNs in terms of model accuracy that is 91.6% but it takes more model testing time. In regard, to model testing speed and size SSqnxt performs better than others (1 second per testing epoch with model size 0.449 MB).

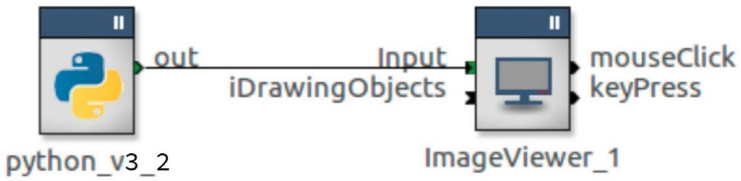


Fig. 9.4. Graphical Interface in RTMaps.

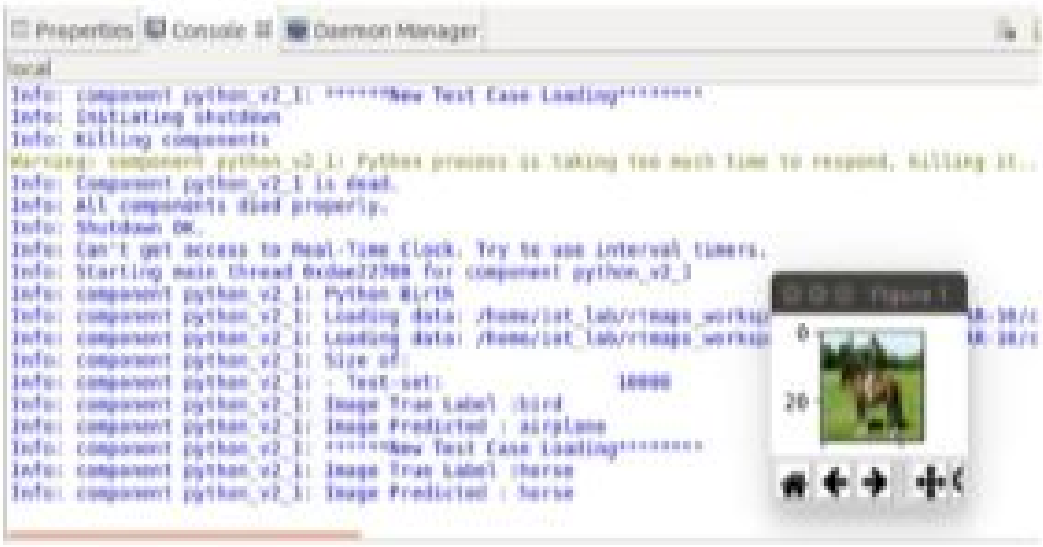
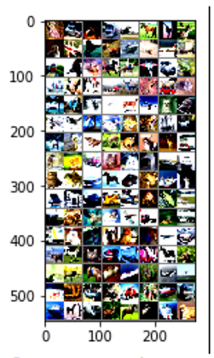


Fig. 9.5. Illustration of Squeezed CNN Classifier Prediction Implemented on BlueBox2.0.

Table 9.1.
BlueBox 2.0 Results

Model	Accuracy%	Model Size(MB)	Model testing speed(sec)
Proposed HPSqNxt-23-1.0x-v1	**91.68**	**2.58**	**2**
Proposed HPSqNxt-06-0.50x-v1	81.84	1.080	1
Proposed SSqNxt-14-1.5x-v1	90.50	8.72	5
Proposed SSqNxt-06-0.575x-v1	81.50	0.449	1
Proposed SSqNxt-09-0.5x-v1	87.30	**0.531**	**1**
SqueezeNext (Baseline)	**87.56**	**2.59**	**2**
Squeezed CNN SqueezeNet Based	**76.32**	**12.9**	-

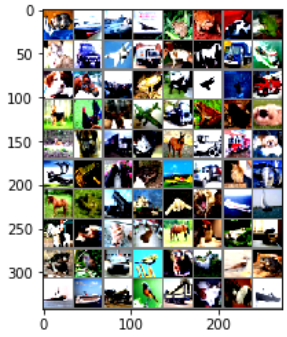
```
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).
```




```
 frog   car  deer truck
Testing Epoch [1/50] Iter[124/125]            Loss:0.0030 Te_Acc: 0.916
Epoch #1 Cost 1s
Testing Epoch [2/50] Iter[124/125]            Loss:0.0030 Te_Acc: 0.916
Epoch #2 Cost 1s
Testing Epoch [3/50] Iter[124/125]            Loss:0.0030 Te_Acc: 0.916
Epoch #3 Cost 1s
Testing Epoch [4/50] Iter[124/125]            Loss:0.0030 Te_Acc: 0.916
Epoch #4 Cost 1s
Testing Epoch [5/50] Iter[124/125]            Loss:0.0030 Te_Acc: 0.916
Epoch #5 Cost 1s
Testing Epoch [6/50] Iter[124/125]            Loss:0.0030 Te_Acc: 0.916
Epoch #6 Cost 1s
Testing Epoch [7/50] Iter[124/125]            Loss:0.0030 Te_Acc: 0.916
Epoch #7 Cost 1s
Testing Epoch [8/50] Iter[124/125]            Loss:0.0030 Te_Acc: 0.916
Epoch #8 Cost 1s
Testing Epoch [9/50] Iter[124/125]            Loss:0.0030 Te_Acc: 0.916
Epoch #9 Cost 1s
Testing Epoch [10/50] Iter[124/125]           Loss:0.0030 Te_Acc: 0.916
```

Fig. 9.6. High Performance SqueezeNext Ground truth & Predicted Images.

```
Epoch #45 Cost 1s
Testing Epoch [46/50] Iter[124/125]            Loss: 0.0033 Te_Acc: 0.905
Epoch #46 Cost 1s
Testing Epoch [47/50] Iter[124/125]            Loss: 0.0033 Te_Acc: 0.905
Epoch #47 Cost 1s
Testing Epoch [48/50] Iter[124/125]            Loss: 0.0033 Te_Acc: 0.905
Epoch #48 Cost 1s
Testing Epoch [49/50] Iter[124/125]            Loss: 0.0033 Te_Acc: 0.905
Epoch #49 Cost 1s
Testing Epoch [50/50] Iter[124/125]            Loss: 0.0033 Te_Acc: 0.905
Epoch #50 Cost 1s
Best Cost: 1s
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).
```




```
GroundTruth:    cat  ship  ship plane
Predicted:    cat  ship  ship plane
Accuracy of plane : 96 %
Accuracy of   car : 95 %
Accuracy of  bird : 86 %
Accuracy of   cat : 75 %
Accuracy of  deer : 85 %
Accuracy of   dog : 82 %
Accuracy of  frog : 100 %
Accuracy of horse : 88 %
Accuracy of  ship : 96 %
Accuracy of truck : 97 %
```

Fig. 9.7. Shallow SqueezeNext Ground truth & Predicted Images.

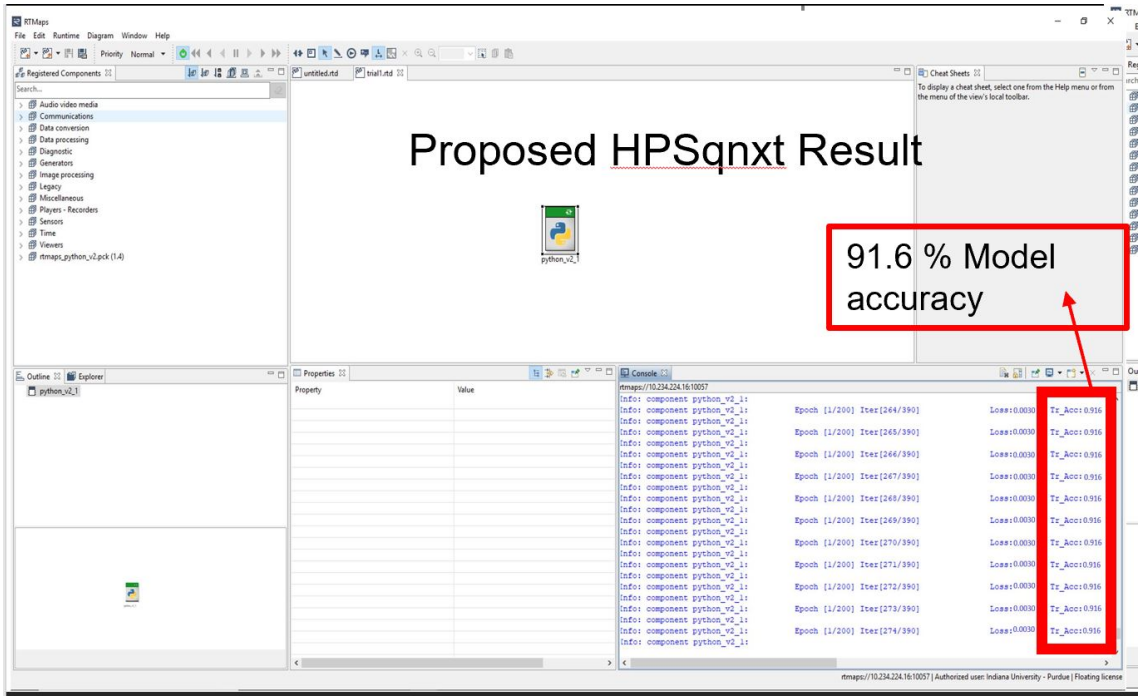


Fig. 9.8. Proposed High Performance SqueezeNext Result on BlueBox2.0.

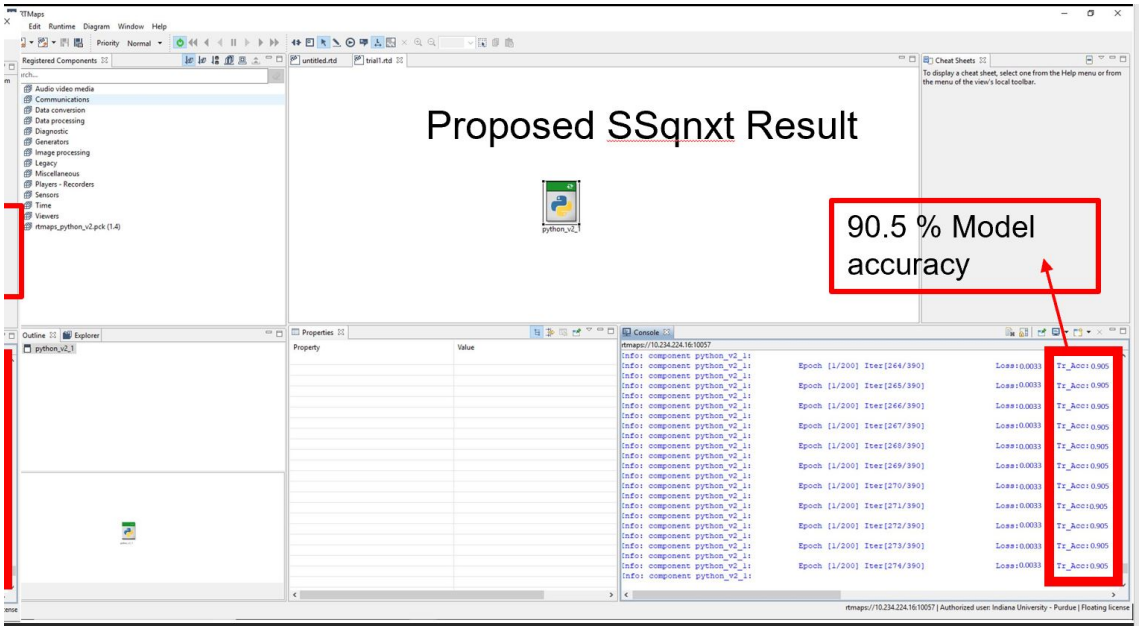


Fig. 9.9. Proposed Shallow SqueezeNext Result on BlueBox2.0.

10. CONCLUSION

The purpose of this thesis is to introduce two new DNN architectures that are High Performance SqueezeNext architecture and Shallow SqueezeNext architecture. In the beginning, we start the thesis study with a introduction to DSE of DNNs. Chapter 2 discusses some important work which lays the foundation and understanding of DSE of DNNs. Then, in the Chapter 3, the related baseline SqueezeNet and SqueezeNext architectures were reviewed. SqueezeNext baseline architecture is the underlying foundation for both of the proposed architectures and briefly explained how SqueezeNext is a more efficient architecture than the SqueezeNet. Then, the general DNN/CNN improvement methods were discussed.

Subsequently, using the acquired knowledge and insights from the previous Chapters, HPSqnxt and SSqnxt architectures are proposed. HPSqnxt has a better model accuracy with decent model speed and size. Then, all the results were discussed in the Chapter 8. Best results achieved for training and testing the proposed High performance squeezenext architecture from scratch on the CIFAR-10 dataset are model accuracy of 92.25%, model size (time cost per epoch) of 0.333 KB, and model speed of 7 seconds. In this thesis, one of the conclusions is that it is important to load the saved model parameters for better accuracy and it is not necessary to save all the model parameters for an efficient model size and model speed.

The choice of optimizers, initialization for convolution [65], batch normalization layers [32], and learning rate decay schedule [80] affects the model's performance. The use of in-place functions [46, 47] improved model size and model speed resulting in the efficient performance of the proposed architecture is one of the key conclusions. Further, it also can be improved by training and testing the model for a specific dataset from scratch without data augmentation [35].

All of the above mentioned benefits make the proposed architecture a flexible DNN architecture considering the tradeoff between the model size, speed and accuracy. HPSqnxt results clearly shows the tradeoff between model's speed, size and accuracy for different resolution and width multipliers. Reducing the depth of the model did not necessary decrease the model accuracy. A shallow model can be trained to mimic a deep model and in this study this version of DNN is Shallow SqueezeNext architecture. Replacing ReLU with ELU-in-place has a good impact on learning and accuracy of the model. Choice of hyper parameters, resolution, and width multipliers are the key in reducing the losses, attaining a better model size, and model accuracy.

In the results, SGD and Adabound optimizers seemed to perform better as compared to other optimizer alternatives for training and testing of the model on CIFAR-10, CIFAR-100 [39] from scratch. The use of average pooling layer instead of max pooling layer along with the use of drop out layer, increases the accuracy with a model speed tradeoff but model size remains unaffected by it. Since the experiments were conducted without data augmentation, using data augmentation [81-83] will likely have a further positive impact on the results.

The proposed Shallow SqueezeNext seems to have better model size and accuracy than the baseline SqueezeNext model trained from scratch on CIFAR-10, CIFAR-100. The proposed architecture with different resolution and width multiplier combinations can be used to create an efficient and flexible CNN model depending on the user-end application. It is expected that with the incredibly small model size of 296MB and model accuracy 82.47%.

Therefore, the proposed Shallow SqueezeNext architecture can be deployed efficiently on ADAS systems/ autonomous systems [41, 42, 75, 77, 85] for potential applications [52, 44, 56, 71] such as image detection, recognition and classification. These results proved that the bottleneck modules could be used to recreate the existing architectures and with fine-tuning and with further DSE of DNN better results with competitive accuracy can be obtained, which is feasible for deployment on embedded systems. In the end, we implement and deploy both architectures on

the BLBX2 by NXP attaining a model accuracy of 91.68% for proposed HPSqnxt Architecture and 81.5% for SSqnxt Architecture with model size of 2.58 MB and 0.449 MB, respectively.

10.1 Summary

The motivation behind [66, 67] this thesis has been the development of DNN architectures designed specifically for real time embedded systems. During this thesis, two DNN architectures where developed using bottleneck modules. Firstly, a compact architecture called High Performance SqueezeNext was developed for the CIFAR-10, CIFAR-100 datasets, was trained and compared to the baseline SqueezeNet and SqueezeNext model. Then, introduces another DNN architecture named Shallow SqueezeNext. Both proposed DNNs were created using compact convolution filters called bottleneck modules. Chapter 8, highlights the results for both of the proposed DNN architectures. Both of the DNNs are flexible, compact & were deployed on an autonomous development platform BLBX2 using RTMaps [43, 90, 91]. Followed by the conclusion of the thesis study. In the end, it discusses the future work within the scope of this thesis study.In short thesis summary is as follows:

- Proposed High Performance SqueezeNext & Shallow SqueezeNext.
- Trained and tested on CIFAR-10 & CIFAR-100.
- Deployed the proposed architectures on BlueBox2.0.
- Proposed SSqnxt Best model size: 196 KB (15x better than SqNet & 13x better than SqNxt).
- Proposed HPSqNxt Best model accuracy: 92.05 % (14 % better than SqNet Baseline & 4.5 % better than SqNxt Baseline).
- Proposed SSqnxt Best model speed: 4 sec/epoch (15 secs better than Sqnxt and equivalent to SqNet).

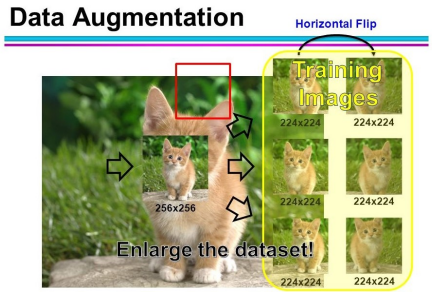
- Proposed HPSqnxt Best BLUEBOX2.0 accuracy: 91.6 % (15.36 % better than Squeezed CNN, 4.12 % better than SqNxt Baseline).
- Proposed SSqnxt Best BLUEBOX2.0 size: 0.531 MB (12.369 MB better than Squeezed CNN, 2.05 MB better than Sqnxt Baseline).

11. FUTURE WORK

General ways to improve the performance of a DNN is mentioned below:

- Analyze errors (bad predictions) in the validation dataset.
- Monitor the activations.
- Monitor the percentage of dead nodes.
- Apply gradient clipping (in particular NLP) to control exploding gradients.
- Shuffle dataset (manually or programmatically).
- Train dataset on large dataset.
- Implement data augmentation.

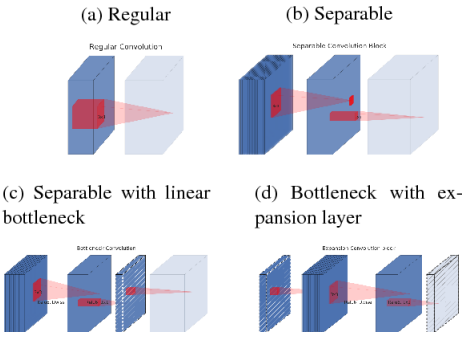
(a) Use a Large Dataset to Train.



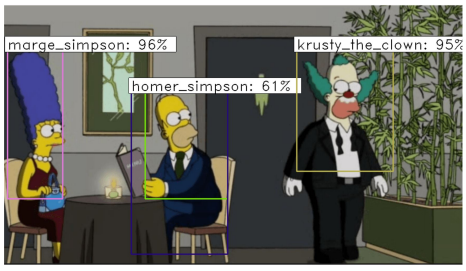
(b) Data Augmentation.

Fig. 11.1. Possible Future Improvements 1.

Firstly, the proposed datasets can be trained from scratch on large datasets such as ImageNet, SVHN, STL-10, etc. After this, the existing model is trained on the smaller datasets [39] which can further improve the image classifier performance i.e.



(a) Use of Inverted Bottleneck Modules Instead of Bottleneck Modules.



(b) Building a Custom Object Detector on the Top of the Proposed Image Classifiers from Scratch.

Fig. 11.2. Possible Future Improvements 2.

model accuracy, model speed and model size. If more data is there, a CNN or DNN model will perform better and fine tuning to continue improving its performance. Fig. 11.3 shows a traditional DL algorithm hits an early data saturation point after which more data does not help in contrast to DNNs.

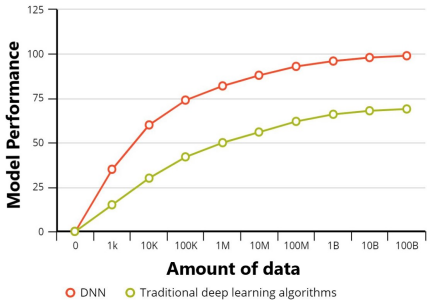


Fig. 11.3. DNNs vs Traditional DL Algorithms.

Secondly, data augmentation [35, 81-83] can be implemented on the top of the proposed architectures to improve DNNs. In the absence of an adequate volume of training data, it is possible to increase the effective size of existing data through the process of data augmentation, which has contributed to significantly improving the performance of deep networks in the domain of image classification. This has the effect

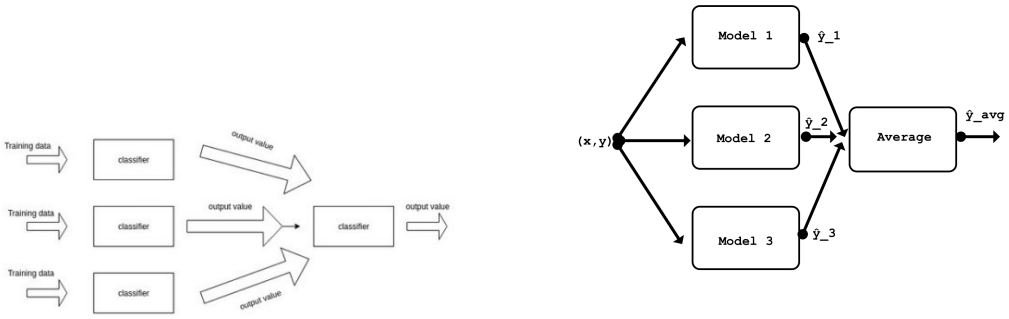
of making the dataset effectively larger, as multiple augmented versions of a single input is fed into the network over the course of training, and also helps the network become robust by forcing it to learn relevant features. However, existing conventional methods of augmenting image input introduces additional computational cost and sometimes requires additional data. Data augmentation for the minority class can synthetically create more training examples for the under represented class.

Thirdly, data ensembles [85] can be used to boost model. In machine learning, ensembles train multiple models and then combine them together to achieve higher performance. Thus, the idea is to train multiple DNN models on the same task and data set. Hence, the results of the models can then be combined through a voting scheme which means the class with the majority of the number of votes wins.

To insure that all of the models are different, random weight initializations and random data augmentation can be used. It is well known that an ensemble is usually significantly more accurate than a single model, due to the use of multiple models and thus approaching the task from different angles. In real-world applications, especially challenges or competitions, almost all the top models use ensembles. The general gist of ensembling is shown below in the Fig. 11.5 (a) & (b), where the outputs of 3 different classification models are combined together.

Also, the proposed architectures with group convolutions [72] cannot be deployed on BLBx2 due to limited support by Pytorch community for porting a program from linux x86 architecture to arm64 architecture. In future , with better support for arm64 these proposed architectures can be deployed on edge devices architectures.

Lastly, model pruning [57, 86] can be done to speed up the proposed architectures. More layers means more parameters, and more parameters leads to more computation, memory consumption, and less speed. A high accuracy can be maintained while increasing our speed with the help of model pruning. The idea is that among the many parameters in the network, some are redundant and do not contribute a lot to the output. If the low ranked neurons in the CNN/DNN can be removed it will result in a smaller and faster network. The ranking can be done according to the L1/L2



(a) Data Ensemble Approach 1. (b) Data Ensemble Approach 2.

Fig. 11.4. Data Ensemble Approaches.

mean of neuron weights, their mean activation, the number of times a neuron was not zero on some validation set, and other creative methods. Getting faster/smaller networks is important for running deep learning networks on mobile devices. The most basic way of pruning networks is to simply drop certain convolutional filters. When pruning a network for deployment and application, the network design is of critical importance, so it is always good to use the latest and greatest methods.

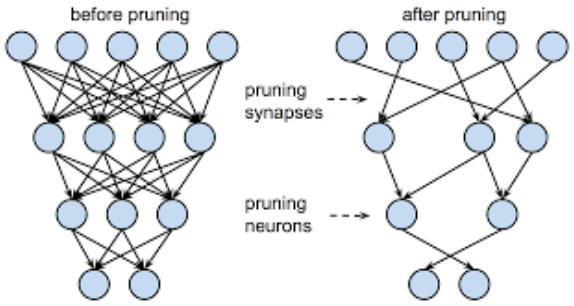


Fig. 11.5. DNN Pruning.

The journey to explore the DSE of DNNs had just started and it will flourish to make room for new AI, machine learning & deep learning technologies.

REFERENCES

REFERENCES

[1] Simonyan, K. and Zisserman, A., 2014. Very deep convolutional networks for large-scale image recognition.(online) arXiv preprint arXiv:1409.1556, (Last Accessed: July, 05 2019).

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

[3] Wu, B., Wan, A., Yue, X. and Keutzer, K., 2018, May. Squeezeseg: Convolutional neural nets with recurrent crf for real-time road-object segmentation from 3d lidar point cloud. In 2018 IEEE International Conference on Robotics and Automation (ICRA) (pp. 1887-1893). IEEE.

[4] Chen, X. and Lawrence Zitnick, C., 2015. Mind's eye: A recurrent visual representation for image caption generation. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 2422-2431).

[5] Iandola, F.N., Han, S., Moskewicz, M.W., Ashraf, K., Dally, W.J. and Keutzer, K., 2016. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and¡ 0.5 MB model size. (online) arXiv preprint arXiv:1602.07360, (Last Accessed: July, 05 2019).

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

[7] Howard, A.G., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., Andreetto, M. and Adam, H., 2017. Mobilenets: Efficient convolutional neural networks for mobile vision applications. (online) arXiv preprint arXiv:1704.04861, (Last Accessed: July, 06 2019).

[8] Sandler, M., Howard, A., Zhu, M., Zhmoginov, A. and Chen, L.C., 2018. Mobilenetv2: Inverted residuals and linear bottlenecks. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (pp. 4510-4520).

[9] Huang, Y., Cheng, Y., Chen, D., Lee, H., Ngiam, J., Le, Q.V. and Chen, Z., 2018. Gpipe: Efficient training of giant neural networks using pipeline parallelism. (online) arXiv preprint arXiv:1811.06965, (Last Accessed: July, 05 2019).

[10] Zhao, B., Feng, J., Wu, X. and Yan, S., 2017. A survey on deep learning-based fine-grained object classification and semantic segmentation. International Journal of Automation and Computing, 14(2), pp.119-135.

[11] Reed, S., Akata, Z., Yan, X., Logeswaran, L., Schiele, B. and Lee, H., 2016. Generative adversarial text to image synthesis. (online) arXiv preprint arXiv:1605.05396, (Last Accessed: July, 05 2019).

[12] Man, D. and Vision, A., 1982. A computational investigation into the human representation and processing of visual information. (online) http://www.contrib.andrew.cmu.edu/~kk3n/80-300/marr2.pdf, (Last Accessed: July, 05 2019).

[13] Li, H., Bhargava, M., Whatmough, P.N. and Wong, H.S.P., 2019, June. On-Chip Memory Technology Design Space Explorations for Mobile Deep Neural Network Accelerators. In Proceedings of the 56th Annual Design Automation Conference 2019 (p. 131). ACM.

[14] McCulloch, W.S. and Pitts, W., 1943. A logical calculus of the ideas immanent in nervous activity. The bulletin of mathematical biophysics, 5(4), pp.115-133.

[15] Hubel, D.H. and Wiesel, T.N., 1959. Receptive fields of single neurones in the cat's striate cortex. The Journal of physiology, 148(3), pp.574-591.

[16] Widrow, B. and Hoff, M.E., 1962. Associative storage and retrieval of digital information in networks of adaptive neurons. In Biological Prototypes and Synthetic Systems (pp. 160-160). Springer, Boston, MA.

[17] Lawrence, R, 1961. Doctor of P.E., Machine Perception of Three-Dimensional, so lids (Doctoral dissertation, Massachusetts Institute of Technology). (online) https://www.researchgate.net/profile/Lawrence_Roberts/publication/220695992_Machine_Perception_of_Three-Dimensional_Solids/links/546d0adc0cf26e95bc3cac04/Machine-Perception-of-Three-Dimensional-Solids.pdf, (Last Accessed: July, 05 2019).

[18] Hopfield, J.J., 1982. Neural networks and physical systems with emergent collective computational abilities. Proceedings of the national academy of sciences, 79(8), pp.2554-2558.

[19] E. Real, A. Aggarwal, Y. Huang, and Q. V. Le. Regularized evolution for image classifier architecture search. (online) arXiv preprint arXiv:1802.01548, 2018, (Last Accessed: July, 06 2019).

[20] Negnevitsky, M., 1970. The History Of Artificial Intelligence Or From The. WIT Transactions on Information and Communication Technologies, 19.

[21] LeCun, Y., Bottou, L., Bengio, Y. and Haffner, P., 1998. Gradient-based learning applied to document recognition. Proceedings of the IEEE, 86(11), pp.2278-2324.

[22] Leung, T. and Malik, J., 1998, June. Contour continuity in region based image segmentation. In European Conference on Computer Vision (pp. 544-559). Springer, Berlin, Heidelberg.

[23] Viola, P. and Jones, M.J., 2004. Robust real-time face detection. International journal of computer vision, 57(2), pp.137-154.

[24] Felzenszwalb, P.F., McAllester, D.A. and Ramanan, D., 2008, June. A discriminatively trained, multiscale, deformable part model. In Cvpr (Vol. 2, No. 6, p. 7).

[25] Krizhevsky, A., Sutskever, I. and Hinton, G.E., 2012. Imagenet classification with deep convolutional neural networks. In Advances in neural information processing systems (pp. 1097-1105).

[26] Zeiler, M.D. and Fergus, R., 2014, September. Visualizing and understanding convolutional networks. In European conference on computer vision (pp. 818-833). Springer, Cham.

[27] He, K., Zhang, X., Ren, S. and Sun, J., 2016. Deep residual learning for image recognition. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 770-778).

[28] Huang, G., Liu, Z., Van Der Maaten, L. and Weinberger, K.Q., 2017. Densely connected convolutional networks. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 4700-4708).

[29] Chetlur, S., Woolley, C., Vandermersch, P., Cohen, J., Tran, J., Catanzaro, B. and Shelhamer, E., 2014. cudnn: Efficient primitives for deep learning. (online) arXiv preprint arXiv:1410.0759, (Last Accessed: July, 05 2019).

[30] Denton, E.L., Zaremba, W., Bruna, J., LeCun, Y. and Fergus, R., 2014. Exploiting linear structure within convolutional networks for efficient evaluation. In Advances in neural information processing systems (pp. 1269-1277).

[31] He, K. and Sun, J., 2015. Convolutional neural networks at constrained time cost. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 5353-5360).

[32] Ioffe, S. and Szegedy, C., 2015. Batch normalization: Accelerating deep network training by reducing internal covariate shift. (online) arXiv preprint arXiv:1502.03167, (Last Accessed: July, 05 2019).

[33] Ludermir, T.B., Yamazaki, A. and Zanchettin, C., 2006. An optimization methodology for neural network weights and architectures. IEEE Transactions on Neural Networks, 17(6), pp.1452-1459.

[34] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I. and Salakhutdinov, R., 2014. Dropout: a simple way to prevent neural networks from overfitting. The journal of machine learning research, 15(1), pp.1929-1958.

[35] Stanley, K.O. and Miikkulainen, R., 2002. Evolving neural networks through augmenting topologies. Evolutionary computation, 10(2), pp.99-127.

[36] Courbariaux, M., Bengio, Y. and David, J.P., 2014. Training deep neural networks with low precision multiplications. (online) arXiv preprint arXiv:1412.7024, (Last Accessed: July, 05 2019).

[37] Sindhwani, V., Sainath, T. and Kumar, S., 2015. Structured transforms for small-footprint deep learning. In Advances in Neural Information Processing Systems (pp. 3088-3096).

[38] Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J. and Wojna, Z., 2016. Rethinking the inception architecture for computer vision. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 2818-2826).

[39] Krizhevsky, A., Nair, V. and Hinton, G., 2009. Cifar-10 and cifar-100 datasets. (online) https://www.cs.toronto.edu/kriz/cifar.html,6, (Last Accessed: July, 05 2019).

[40] Yang, Z., Moczulski, M., Denil, M., de Freitas, N., Smola, A., Song, L. and Wang, Z., 2015. Deep fried convnets. In Proceedings of the IEEE International Conference on Computer Vision (pp. 1476-1483).

[41] Ashraf, K., Wu, B., Iandola, F.N., Moskewicz, M.W. and Keutzer, K., 2016. Shallow networks for high-accuracy road object-detection. (online) arXiv preprint arXiv:1606.01561, (Last Accessed: July, 05 2019).

[42] Erofei, A.A., Drua, C.F. and Cleanu, C.D., 2018, May. Embedded Solutions for Deep Neural Networks Implementation. In 2018 IEEE 12th International Symposium on Applied Computational Intelligence and Informatics (SACI) (pp. 000425-000430). IEEE.

[43] Venkitachalam, S., Manghat, S.K., Gaikwad, A.S., Ravi, N., Bhamidi, S.B.S. and El-Sharkawy, M., 2018. Realtime Applications with RTMaps and Bluebox 2.0. In Proceedings on the International Conference on Artificial Intelligence (ICAI) (pp. 137-140). The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp).

[44] Niu, X.X. and Suen, C.Y., 2012. A novel hybrid CNNSVM classifier for recognizing handwritten digits. Pattern Recognition, 45(4), pp.1318-1325.

[45] Luo, L., Xiong, Y., Liu, Y. and Sun, X., 2019. Adaptive gradient methods with dynamic bound of learning rate. (online) arXiv preprint arXiv:1902.09843, (Last Accessed: July, 05 2019).

[46] Clevert, D.A., Unterthiner, T. and Hochreiter, S., 2015. Fast and accurate deep network learning by exponential linear units (elus). (online) arXiv preprint arXiv:1511.07289, (Last Accessed: July, 05 2019).

[47] Shah, A., Kadam, E., Shah, H., Shinde, S. and Shingade, S., 2016. Deep residual networks with exponential linear unit. (online) arXiv preprint arXiv:1604.04112, (Last Accessed: July, 05 2019).

[48] Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R., Guadarrama, S. and Darrell, T., 2014, November. Caffe: Convolutional architecture for fast feature embedding. In Proceedings of the 22nd ACM international conference on Multimedia (pp. 675-678). ACM.

[49] Radford, A., Metz, L. and Chintala, S., 2015. Unsupervised representation learning with deep convolutional generative adversarial networks. (online) arXiv preprint arXiv:1511.06434, (Last Accessed: July, 05 2019).

[50] Zilly, J.G., Srivastava, R.K., Koutnk, J. and Schmidhuber, J., 2017, August. Recurrent highway networks. In Proceedings of the 34th International Conference on Machine Learning-Volume 70 (pp. 4189-4198). JMLR. org.

[51] Badrinarayanan, V., Handa, A. and Cipolla, R., 2015. Segnet: A deep convolutional encoder-decoder architecture for robust semantic pixel-wise labelling. (online) arXiv preprint arXiv:1505.07293, (Last Accessed: July, 05 2019).

[52] Ruder, S., 2016. An overview of gradient descent optimization algorithms. (online) arXiv preprint arXiv:1609.04747, (Last Accessed: July, 05 2019).

[53] Nooka, S.P., Chennupati, S., Veerabhadra, K., Sah, S. and Ptucha, R., 2016, December. Adaptive hierarchical classification networks. In 2016 23rd International Conference on Pattern Recognition (ICPR) (pp. 3578-3583). IEEE.

[54] Chennupati, S., Sah, S., Nooka, S. and Ptucha, R., 2016. Hierarchical decomposition of large deep networks. Electronic Imaging, 2016(19), pp.1-6.

[55] Oruganti, R.M., Sah, S., Pillai, S. and Ptucha, R., 2016, September. Image description through fusion based recurrent multi-modal learning. In 2016 IEEE International Conference on Image Processing (ICIP) (pp. 3613-3617). IEEE.

[56] Han, S., Mao, H. and Dally, W.J., 2015. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. (online) arXiv preprint arXiv:1510.00149, (Last Accessed: July, 05 2019).

[57] Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G.S., Davis, A., Dean, J., Devin, M. and Ghemawat, S., 2016. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. (online) arXiv preprint arXiv:1603.04467, (Last Accessed: July, 05 2019).

[58] Paszke, A., Gross, S., Chintala, S. and Chanan, G., 2017. Pytorch. Computer software. Vers. 0.3, 1.

[59] Team, P.C., 2017. Pytorch: Tensors and dynamic neural networks in python with strong GPU acceleration, (online) `https://github.com/pytorch/pytorch`, (Last Accessed: July, 02 2019).

[60] Kostrikov, I., 2018. Pytorch implementations of reinforcement learning algorithms, (GitHub repository) `https://github.com/ikostrikov/pytorch-a3c`, (Last Accessed: July, 02 2019).

[61] Chen, L.C., Papandreou, G., Kokkinos, I., Murphy, K. and Yuille, A.L., 2017. Deeplab: Semantic image segmentation with deep convolutional nets, atrous convolution, and fully connected crfs. IEEE transactions on pattern analysis and machine intelligence, 40(4), pp.834-848.

[62] Redmon, J., Divvala, S., Girshick, R. and Farhadi, A., 2016. You only look once: Unified, real-time object detection. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 779-788).

[63] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, . and Polosukhin, I., 2017. Attention is all you need. In Advances in neural information processing systems (pp. 5998-6008).

[64] Karpathy, A., 2017. CS231n Convolutional Neural Networks for Visual Recognition. 2016. (online) `http://cs231n.github.io,50`, (Last Accessed: July, 05 2019).

[65] Wu, L.Y., Fisch, A., Chopra, S., Adams, K., Bordes, A. and Weston, J., 2018, April. Starspace: Embed all the things!. In Thirty-Second AAAI Conference on Artificial Intelligence, (online) `https://www.aaai.org/ocs/index.php/AAAI/AAAI18/paper/view/16998/16114`, (Last Accessed: July, 06 2019).

[66] Reed, S., Akata, Z., Lee, H. and Schiele, B., 2016. Learning deep representations of fine-grained visual descriptions. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (pp. 49-58).

[67] Schroff, F., Kalenichenko, D. and Philbin, J., 2015. Facenet: A unified embedding for face recognition and clustering. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 815-823).

[68] Hochreiter, S. and Schmidhuber, J., 1997. Long short-term memory. Neural computation, 9(8), pp.1735-1780, (online) https://www.mitpressjournals.org/doi/pdf/10.1162/neco.1997.9.8.1735, (Last Accessed: July, 06 2019).

[69] Zhang, T., Qi, G.J., Xiao, B. and Wang, J., 2017. Interleaved group convolutions. In Proceedings of the IEEE International Conference on Computer Vision (pp. 4373-4382), (online) http://openaccess.thecvf.com/content_ICCV_2017/papers/Zhang_Interleaved_Group_Convolutions_ICCV_2017_paper.pdf, (Last Accessed: July, 06 2019).

[70] Kingma, D.P. and Ba, J., 2014. Adam: A method for stochastic optimization. (online) arXiv preprint arXiv:1412.6980, (Last Accessed: July, 05 2019).

[71] Cubuk, E.D., Zoph, B., Mane, D., Vasudevan, V. and Le, Q.V., 2018. Autoaugment: Learning augmentation policies from data. (online) arXiv preprint arXiv:1805.09501, (Last Accessed: July, 05 2019).

[72] Bahdanau, D., Cho, K. and Bengio, Y., 2014. Neural machine translation by jointly learning to align and translate. (online) arXiv preprint arXiv:1409.0473, (Last Accessed: July, 05 2019).

[73] Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A. and Bengio, Y., 2014. Generative adversarial nets. In Advances in neural information processing systems (pp. 2672-2680), (online) http://papers.nips.cc/paper/5423-generative-adversarial-nets.pdf, (Last Accessed: July, 04 2019).

[74] Nguyen, A., Clune, J., Bengio, Y., Dosovitskiy, A. and Yosinski, J., 2017. Plug & play generative networks: Conditional iterative generation of images in latent space. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (pp. 4467-4477), (online) http://openaccess.thecvf.com/content_cvpr_2017/papers/Nguyen_Plug__Play_CVPR_2017_paper.pdf, (Last Accessed: July, 06 2019).

[75] Gordo, A., Almazan, J., Revaud, J. and Larlus, D., 2017. End-to-end learning of deep visual representations for image retrieval. International Journal of Computer Vision, 124(2), pp.237-254.

[76] Donahue, J., Anne Hendricks, L., Guadarrama, S., Rohrbach, M., Venugopalan, S., Saenko, K. and Darrell, T., 2015. Long-term recurrent convolutional networks for visual recognition and description. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 2625-2634), (online) http://openaccess.thecvf.com/content_cvpr_2015/papers/Donahue_Long-Term_Recurrent_Convolutional_2015_CVPR_paper.pdf, (Last Accessed: July, 05 2019).

[77] Ge, R., Kakade, S.M., Kidambi, R. and Netrapalli, P., 2018. Rethinking learning rate schedules for stochastic optimization, (online) https://openreview.net/pdf?id=HJePy3RcF7, (Last Accessed: July, 06 2019).

[78] Xie, Q., Dai, Z., Hovy, E., Luong, M.T. and Le, Q.V., 2019. Unsupervised data augmentation. (online) arXiv preprint arXiv:1904.12848, (Last Accessed: July, 05 2019).

[79] Van Dyk, D.A. and Meng, X.L., 2001. The art of data augmentation. Journal of Computational and Graphical Statistics, 10(1), pp.1-50.

[80] Wong, S.C., Gatt, A., Stamatescu, V. and McDonnell, M.D., 2016, November. Understanding data augmentation for classification: when to warp?. In 2016 international conference on digital image computing: techniques and applications (DICTA) (pp. 1-6). IEEE.

[81] Zhang, J., Huang, M., Jin, X. and Li, X., 2017. A real-time chinese traffic sign detection algorithm based on modified YOLOv2. Algorithms, 10(4), p.127.

[82] Grell, G.A. and Dvnyi, D., 2002. A generalized approach to parameterizing convection combining ensemble and data assimilation techniques. Geophysical Research Letters, 29(14), pp.38-1.

[83] Gaikwad, A.S., 2018. Pruning Convolution Neural Network (SqueezeNet) for Efficient Hardware Deployment (MSECE Thesis, Purdue University, Indianapolis).

[84] Pathak, D. and El-Sharkawy, M., 2019, January. Architecturally Compressed CNN: An Embedded Realtime Classifier (NXP Bluebox2. 0 with RTMaps). In 2019 IEEE 9th Annual Computing and Communication Workshop and Conference (CCWC) (pp. 0331-0336). IEEE.

[85] Duggal, Jayan Kant and El-Sharkawy, Mohamed , High Performance SqueezeNext for CIFAR-10,2019 National Aerospace
& Electronics Conference, July 15-18, 2019, Dayton, Ohio, USA (accepted).

[86] Duggal, Jayan Kant and El-Sharkawy, Mohamed , Shallow SqueezeNext and Efficient DNN, IEEE International Conference on Vehicular Electronics and Safety (ICVES19), September 4-6, 2019, Cairo, Egypt (accepted).

[87] Duggal, Jayan Kant and El-Sharkawy, Mohamed , High Performance SqueezeNext,2019 IEEE Journal on Transactions on Pattern Analysis and Machine Intelligence, July, 2019 (submitted).

[88] Duggal, Jayan Kant and El-Sharkawy, Mohamed , Shallow SqueezeNext,2019 IEEE Journal on Transactions on Pattern Analysis and Machine Intelligence, August, 2019 (to be submitted).

[89] Harley, A.W., 2015, December. An interactive node-link visualization of convolutional neural networks. In International Symposium on Visual Computing (pp. 867-877). Springer, Cham, (online) http://scs.ryerson.ca/~aharley/vis/harley_vis_isvc15.pdf, (Last Accessed: July, 05 2019).

[90] Nwankpa, C., Ijomah, W., Gachagan, A. and Marshall, S., 2018. Activation functions: Comparison of trends in practice and research for deep learning. (online) arXiv preprint arXiv:1811.03378, (Last Accessed: July, 05 2019).