ALGORITHMIC OPTIMIZATION OF FIRST CONVOLUTION LAYER IN CNNS FOR HARDWARE ACCELERATOR DESIGN

by

Ramachandra Vikas Chamarthi

A thesis submitted to the faculty of
The University of North Carolina at Charlotte
in partial fulfillment of the requirements
for the degree of Master of Science in
Electrical Engineering

Charlotte

2019

Approved by:

_____
Dr. Hamed Tabkhi

_____
Dr. James M Conrad

_____
Dr. Andrew R Willis

ABSTRACT

RAMACHANDRA VIKAS CHAMARTHI. Algorithmic Optimization of First Convolution Layer in CNNs for Hardware Accelerator Design. (Under the direction of DR. HAMED TABKHI)

This thesis proposes "1D Convolution replacement layer ", a novel optimization for first convolution layer in CNN. This optimization enables edge friendly streaming accelerator design with a minimum drop in accuracy. This optimization reduces the number of convolution parameters in the first convolution layers of CNN, reducing the number of multiplications performed in convolution operation. In CNNs first convolution is the most memory and compute intensive as the first layer operates on input. In a streaming accelerator design, the first layer operates on streaming data, the complexity of operations and memory demand of the first layer will proportionally affect the latency of complete accelerator design. Using 1D Convolution replacement in a CNN on a N x N convolution layer after 1D replacement number of operations in each convolution window gets reduced by N times. To show the effect of 1D convolution in accelerators, streaming accelerator design for SqueezeNet is compared with 1D-SqueezeNet, SqueezeNet with 1D convolution replacement in the first layer in [1] is discussed. 1D replacement enabled edge friendly design reducing the dynamic power consumption by 7.3X, with 0.6% drop in accuracy in SqueezeNet real-time edge accelerator. 1D Convolution replacement is trained on a variety of CNN networks using various datasets, and are compared against original CNN networks to understand the scope and applicability of 1D Convolution replacement layer.

ACKNOWLEDGEMENTS

I want to acknowledge my committee chair Dr. Hamed Tabkhi and TeCSAR research group for the enormous support and design of custom accelerator on real hardware helping me to show the advantage of my algorithmic optimization of CNN's. I would also like to acknowledge Proscia Inc, Philadelphia for providing industrial training helping me experiment on varieties of data enabling me to widen the understanding of Deep Learning.

DEDICATION

This thesis is dedicated to my advisor, Dr. Hamed Tabkhi for constant support and guidance. Also to my parents for the unmeasurable moral support and encouragement.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

1D One Dimensional

2D Two Dimensional

BRAM Block Random Access Memory

CLB Configurable Logic Block

CNN Convolution Neural Networks

Conv Convolution Layer

CPE Convolution Processing Engine

FC Fully Connected Layer or Linear Layer

FPS Frames Per Second

GEMM General Matrix Multiplication

GPU Graphics Processing Unit

ILSVRC Imagenet Large Scale Visualization Competition

im2col Image to column

LUT Logic Utilization

MAC Multiplication and Accumulation

Maxpool Maxpooling Layer

MNIST Modified National Institute of Standards and Technology database

PPE Pooling Processing Engine

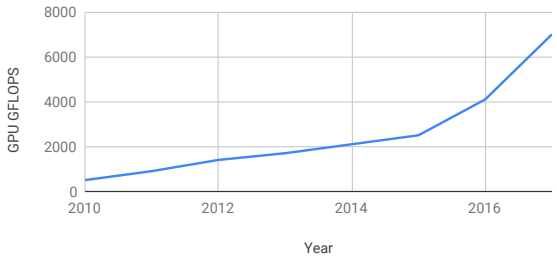
CHAPTER 1: INTRODUCTION

Convolution Neural Networks (CNNs) are widely used in many fields to solve many machine learning problems [2] utilizing big data and GPUs [3]. Even though the concept of neural networks is old, GPUs provided computation power to train the CNN with millions of neurons connecting the input to output. Majority of complex applications using CNNs are image-based [2] and all the CNN's utilize convolution layer, Convolution layer use filters to detect [4] the presence of filter or a feature in the given input. The complexity of features to be extracted vary in multiple hierarchies of Convolution Neural Networks are widely used in many fields to solve many machine learning problems [2] utilizing big data and GPUs [3]. Even though the concept of neural networks is old, GPUs provided computation power to train the CNN with millions of neurons connecting the input to output. Majority of complex applications using CNNs are image-based [2] and all the CNNs utilize convolution layer, Convolution layer use filters to detect [4] the presence of filter or a feature in the given input. The complexity of features to be extracted vary in multiple hierarchies of features ranging from lower level features like simple strides and shapes to complex features like ears, eyes, and wheels. To the further layers activation outputs from the previous layer are given as inputs, and pooled activations are used to detect complex features, and further convolution layers are used to detect the patterns of activation's which represent higher-level features. For example, feature presence of eyes, ear, and mouth would represent higher feature face in the image.

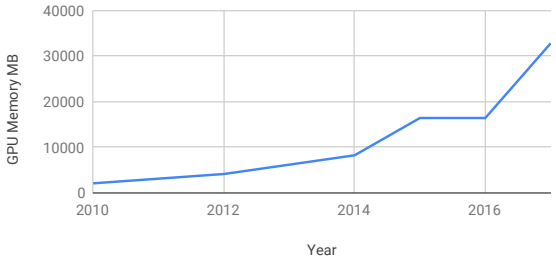
In recent years GPUs have become more efficient [5],[6], Since Nvidia STG-2000 in 1995, with 4MB Memory to Nvidia V100 with 32GB Memory [5].

As shown in Fig.1.1 GPU's performance also increased from 500 GFLOPS to 7000

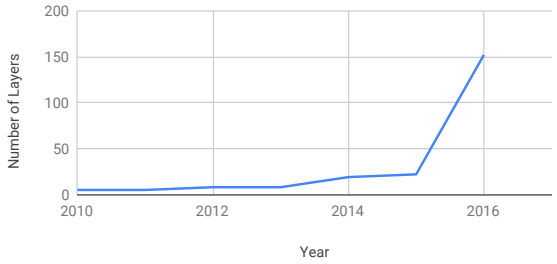
GFLOPS in recent Volta architecture [5]. This advancement provided researchers ability to train algorithms with larger depth with minimum time.In [7] Resnet50 was trained on Imagenet data in 224 seconds. This advanced CNN's from Lenet [8] in 1998, 5 layer CNN with 60k parameters to latest ResNet-152 [9], 152 layer CNN with 60 Million parameters, reducing the Imagenet top-5 error from 30% to 4%. Even though gaming sector was the major driving force for GPU research, since last two years architecture optimizations in Volta and Turing Architecture were deep learning oriented, in Volta Tensor cores were introduced [5] and in Turing architecture mixed precision computing was introduced [6]. Previous to Volta architecture basic computation blocks were MACs and after Volta architecture Tensor Cores (Matrix multiplication engines) were introduced. Matrix multiplication is the basic computation block for convolution because of GEMM acceleration library [3].



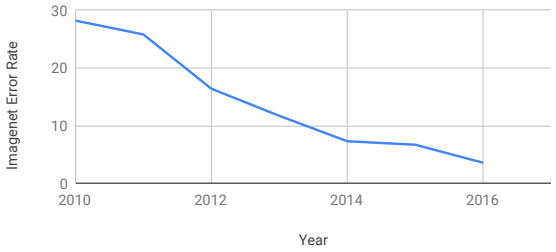
(a) GPU GFLOPS per year



(b) GPU Memory per year



(c) CNN Number of layers per year



(d) GPU Imagenet Top-5 error % per year

Figure 1.1: Advancements in GPU's and CNN's from 2012 to 2016

In general, CNNs are multiple convolution blocks, each convolution block generally contain a Convolution layer , activation layer, and pooling layer [4],[10],[11]. Multi-

ple convolution blocks are placed one after the other reducing the input in to 1000 values. As layers progress towards the network end, the number of convolution filters increase from 64 to 1000 filters, producing final output of 1 x 1 x 1000 representing 1000 labels. In convolution layer convolution filter is applied with stride on to input, similar to moving window fashion. GPUs are throughput oriented machines, and only with embarrassingly parallel data utilization is maximized [3]. Convolution operation includes stride, strided convolution create data dependency between consecutive moving convolution windows restricting the number of threads launchable in parallel. In most of GPU based deep learning environments GEMM library is used convert image into the column, with im2col transformation and leverage large GPU memory available to create multiple copies of data common between strides and solve the stride dependant data dependency [3].

In real time accelerator designs for edge inference [1], [12], Input tiles are streamed, and accelerator needs to process at 60fps or 30fps based on requirement, and minimum latency model can be designed only by allocation layer resources such that each stage can be pipe-lined to process streaming inputs. As first convolution layer operates on streaming input, the latency of first convolution layer effects the operation of following layers [1] as processing rate of first layer effects the rate at which the next layers are processes as for further layers data input is not streamed but is the output of the first layer. While designing datapaths in custom hardware accelerators to match per layer computation, design resources and power requirements scale proportionally to computational complexity.

1.1 Motivation

Even though GPUs are ideal for training CNNs, for deployment of CNNs during application in the real world custom hardware designing is required to match the application latency and power requirements. Optimizing algorithms to enable edge friendly models with minimum latency and lower power consumption will be

important. Proposed 1D Convolution replacement approach enables a lower power consumption hardware design.

This thesis proposes an edge friendly 1D convolution replacement optimization to enable an edge friendly accelerator design with dynamic power consumption 7.3 times less than 2D convolution accelerator design of real-time streaming SqueezeNet accelerator. In this thesis, various networks were trained with 1D convolution replacement algorithm on various types of CNN architectures and different image classification datasets like MNIST, CIFAR10, CIFAR100, and Imagenet.

The organization of the remaining section is as follows: Background section 2 contains information CNN's and accelerators basics, and Analysis section 3 follows the background section with CNN analysis, and Approach section 4 explaining the 1D Convolution replacement approach, and Results section 5 reporting 1D replacement results on various CNNs and architectural results showing the advantage of approach.

CHAPTER 2: BACKGROUND

As stated in Introduction, CNNs are stacked blocks of feature extractors [4],[10] extracting various hierarchies of patterns with multiple filters with convolution operation and pooled activations of feature presence obtained from the convolution operation. Each convolution block usually consists of a Convolution layer, an Activation layer, and a Pooling layer. The number of blocks represents the depth of the network representing the complexity of the network concerning the number of neural connections connecting the inputs to the outputs.

Even though convolutional neural networks are old, application of them on to image domain started only after the advent of GPUs, throughput oriented machines providing necessary computational resources for computing the enormous number of multiplications and accumulations. General network training involves a forward pass and a backward pass. Forward pass involves computation of all the convolution blocks and production of required output. Backward pass involves computation of loss and gradients for each weight, which are filters of convolution layers used to extract features and update the weights such that weights tend to become the required feature extractors.

Forward pass involves computation of convolution blocks. These computations translate as multiplications and accumulations [1]. But in the advent of GPUs with large memory, the large number of cores [5],[6], and large register banks enabled the ability to launch thousands of threads without any data dependency making GPUs perfect use case for Convolutional Networks exploiting the data parallelism in computation. As in CNNs, all the operations are performed on parallel as blocks of data. As GPU's are throughput oriented, for maximum utilization of cores, algorithms, and

architectures are even optimized to optimize the convolution operation[3].

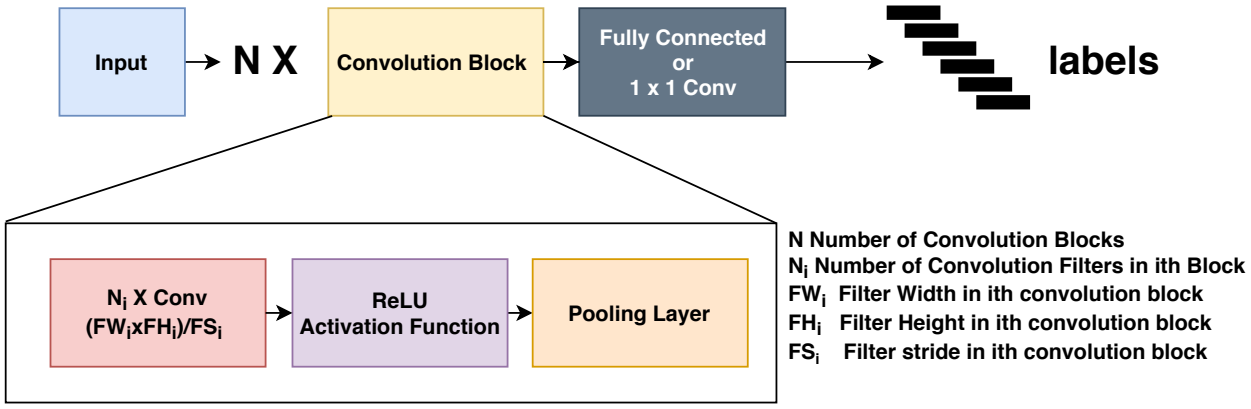


Figure 2.1: General CNN process image

2.1 Convolution Neural Networks (CNN)

CNNs are being widely used to solve a variety of problems in many domains like speech, Image, Discrete data. Many current advances are focused on Image domain [13],[2],[14],[15].

2.1.1 Regular CNN

Since LeNet [4], to recent Resnet Architecture, CNN's have advanced from 5 layers to 152 layers [9]. The increasing number of the trainable parameter from hundreds of thousands to billions. Even GPU architectures evolved since G80 architecture to the recent Turing [6] architecture to fit a large amount of data.

Various networks are trained using 1D replacement from scratch on various datasets. Fig-2.3 and Fig-2.2 are the network architectures of SqueezeNet and Alexnet. Alexnet [10] is the 2012 Imagenet [16] challenge winner and SqueezeNet is network introduced after Alexnet. In Alexnet multiple convolution blocks followed by a Fully connected layer were used reduce the input to 1 x 1 x 1000 dimensions associating to 1000 labels. SqueezeNet [17] achieved Alexnet level accuracy with 50x fewer parameters by utilizing complex features obtained by fire modules as shown in Fig-2.2, layers concatenating multiple features from multi-sized convolution layers and squeezing them

with 1x1 convolutions and expanding concatenating the outputs of 1 x 1 and 3 x 3 convolutions. Using fire modules [17] effective depth of network is increased with less number of parameters and also by replacing the final FC layer with 1 x 1 Convolution and average pooling. In further models, ILSVRC 2014 winner Googlenet [18], Inception module was introduced going much deeper in the layer number and also concatenating 1 x 1, 3 x 3 and 5 x 5 convolution outputs. Since Resnet [9] Residual links introduced to bypass the information passing from all layers avoid vanishing of feature solving the problem of vanishing gradients, Allowing them to create a network with 152 layers, Resnet-152. Along with above stated networks ZFNET [19], 2013 ILSVRC winner with minor change of changing first convolution layer from 7 X 7 to 5 x 5 to train better on small inputs, and Capsnet [20] were trained with 1D replacement to evaluate the 1D convolution replacement approach.

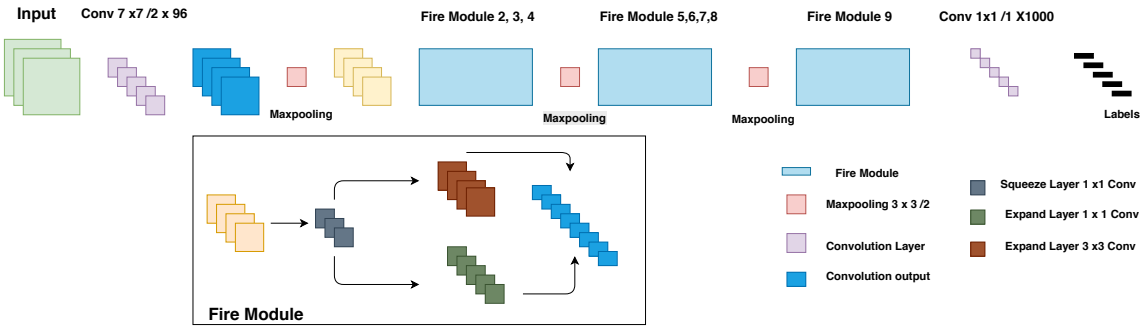


Figure 2.2: SqueezeNet architecture

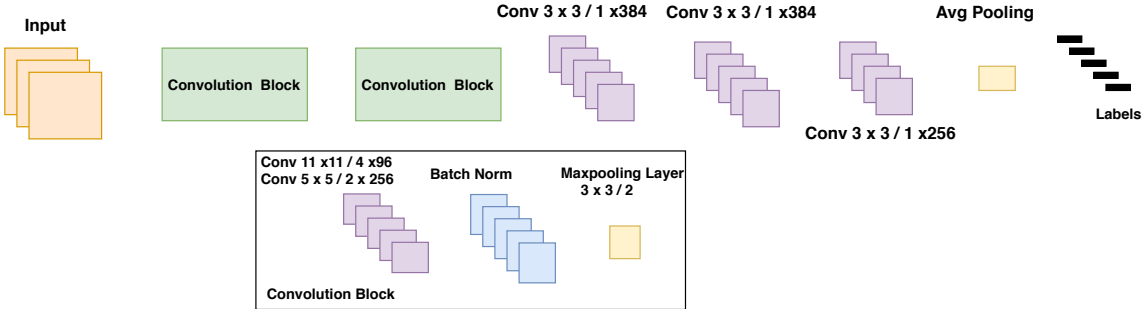


Figure 2.3: Alexnet architecture

2.1.2 Capsnet

In 2018, Capsule Network [20] introduced the dynamic routing, overcoming the major drawback in existing CNN's inability to learn rational invariance because of scalar accumulation of activation outputs with pooling dismissing the positional importance of activations. Fig-2.4 shows the architecture of Capsnet consisting three layers Convolution layer, Primary capsule layer and digits capsule layer. To the first convolution layer inputs are fed, generating lower level features, These feature maps are fed to primary capsule layer, this layer generated multiple combinations of feature maps and generated feature combinations are reshaped to capsules to generate combinations as capsules containing presence and pose information. Thus capsule outputs are reshaped into N X k dimensional vectors, with k being the dimension of the vector in the capsule and N being the number of capsules after reshaping of features into k dimensional capsules.

In Fig-2.4, architecture given in [20] and also used in my experiments on MNIST [8] and CIFAR10 [21] dataset. In case of Capsnet on MNIST dataset has input size of $28 \times 28 \times 1$ fed to first convolution layer with 256 filters of filter size 9 x 9 and stride 1, generating an output of size $20 \times 20 \times 256$. Output of first convolution layer is fed input primary capsule layer with 256 filters of filter size 9×9 and stride 1 to generate capsule outputs of $6 \times 6 \times 256$ $(6 \times 6 \times 8 \times 32)$, i.e $6 \times 6 \times 32$, 8 dimensional capsules, they are reshaped in to $1152(6 \times 6 \times 32) \times 8$ representing 1152 capsules of eight dimensions. These 1152 features are dynamically routed to 10 digit capsules of dimension 16 representing 16 features of each label. In process of dynamic routing a weight matrix of size $1152 \times 8 \times 16$ matrix multiplied on to $1152 \times 1 \times 8$ feature maps and weights are iteratively optimized to route similar low-level features to digit capsules of the higher level process [20].

In Capsnet regular scalar activations are replaced with vector activation encoded with both poses along with its presence. The iterative dynamic routing algorithm

[20] is used to route the lower level feature to higher level features. Dynamic routing uses similarity score, i.e., dot product to find the similar features and iteratively route the closest feature. The Table 3.1 shows the dynamic routing algorithm given in [20] translated to matrix operations.

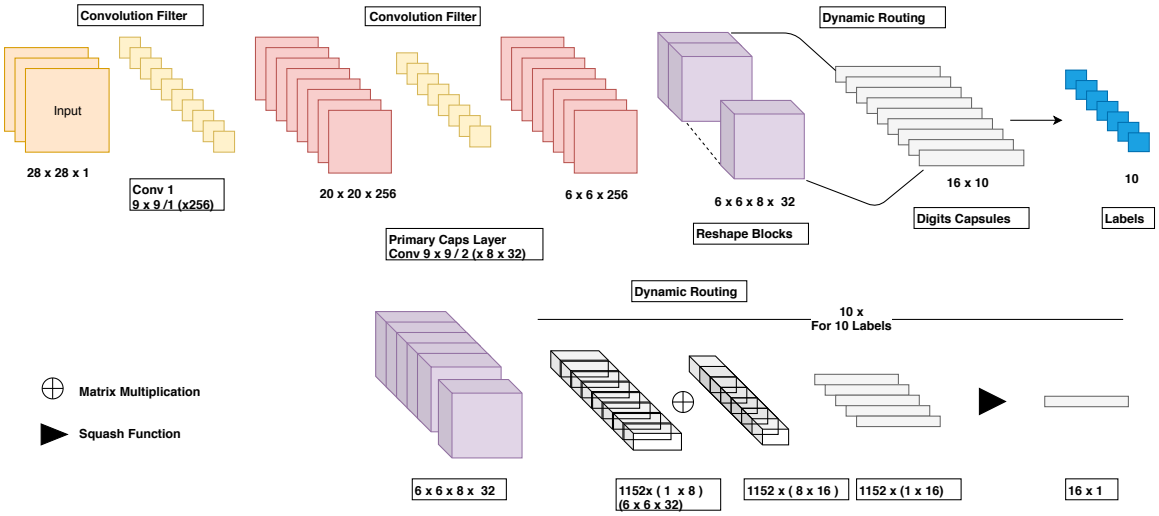


Figure 2.4: Capsnet Architecture

Table 2.1: Dynamic routing algorithms in Capsnet architecture

S No	Routing Algorithm pseudo code
step 0	Routing procedure between lower level capsules u in layer l and
	higher level capsules v in layer l + 1 ui is the output of ith capsule in layer l;vj is the output of jth capsule of layer l+1
step 1	for all capsules i in layer l for all capsule j in layer l + 1 and bij is the coupling coefficient or routing coefficient between ith capsule in layer l and jth capsule in layer l + 1 initialized to zero.
step 2	for number of routings
step 3	normalize b to find the nearest coefficients
step 4	for all capsule layer l + 1 : output for each capsule sj = sigma bij .uj|i
step 5	for all capsule layer l + 1 : output vj = Squash (sj)
step 6	for all capsule layer i in layer l and for all capsules j in layer l + 1 : bij = bij + uji.vj

2.2 CNN Accelerators

Increasing applications for CNN's and the increase of real-time CNN inference requirements and GPU's inefficiency [1] to handle streaming data as cores would stay underutilized because of the insufficient amount of parallelism exploitable with a single image compared to the batch of images [22]. In case of streaming accelerator that processes at a rate of fps or ips, input data continuously stream in and by designing a complete data pipeline from input to maximum output utilization is achieved. Streaming input avoid the requirement of hardware reuse between layers, as next stream of data streams in to occupy and utilize. Thus designing layer specific datapaths for each layer would be Ideal. Fig-2.5 shows the translation of each convolution block into each hardware unit. Thus for multiple convolution blocks, each of hardware block is replaced with block specific design. The data paths for each block is varied according to the number of convolution filters and size of convolution filters. As shown in Fig-2.5 Convolution operation using an array of multipliers and adders

and Activation layer being conditional operation array of comparators [1] are used, pooling is performed using an array of adders and a buffer. As streaming data comes in pixel by pixel, data has to be temporarily stored to compute convolution in 2D line buffers to store data row by roe and process them as the data streams. Size of the 2D line buffer is dependant on the number of values to be processed per convolution window. By reducing the complexity of computation in first convolution layer, i.e reducing the memory accesses and computation cycles of first convolution layer efficient hardware designs would be possible.

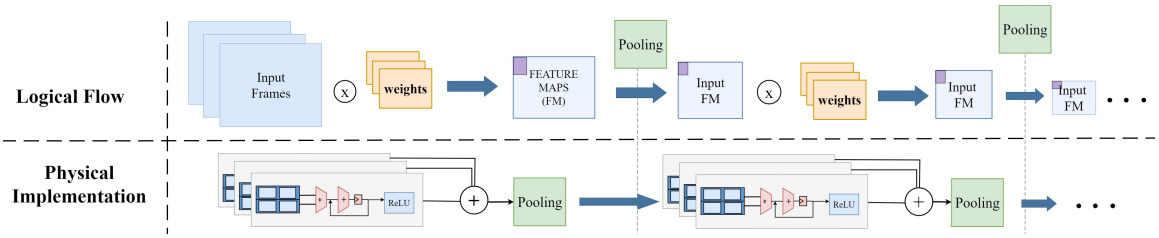


Figure 2.5: Algorithm to Hardware translation

CHAPTER 3: ANALYSIS

This section discusses the analysis of CNNs. Capsnet computational analysis is given a special section as it exhibits a high level of computational complexity than regular CNN and deserves a special section because of its importance and ability to the feature variance.

3.1 GEMM Memory Requirement and Computation

In many Deep learning libraries [23],[24],[25],[25] GEMM convolution is used and input and weights are transformed in to a column of convolution patches using the im2col function, In im2col function, each convolution filter application window is transformed into a single column and respective filter weights are transformed into a row. By this transformation operation of convolution is transformed into matrix multiplication. In the convolution layer with stride, data overlap between the consecutive convolution filter applications when converted into im2col create data redundancy between each column. As shown in Fig-3.1 each convolution patch translates to a single column and as multiple convolution patches overlap common data between them is duplicated to avoid data dependency between two patches [3].

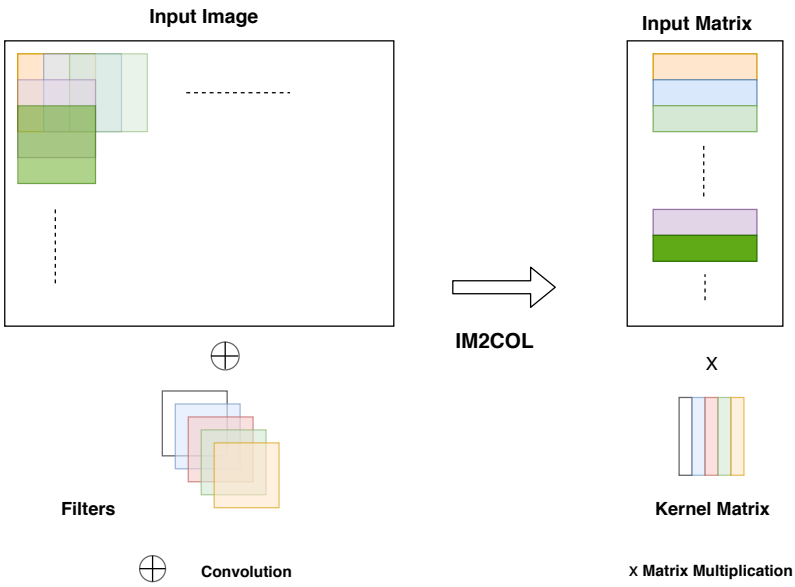


Figure 3.1: GEMM conversion in convolution operation

Using equations 3.1 and 3.2 for a Input of size $W \times H \times C$ being width, height and number of channels of input. For convolution Filter of size F_h x F_w, with strides S_h,S_w and with padding P_h, P_w. Image to column transformed input matrix size i computed as I_r and I_c, Ir being the number of rows and I_c being the number of columns. Size of memory after im2col transformation would be $I_r \times I_c$. In case of convolution number of multiplication would be equal to $I_r \times I_c$ and number of additions or accumulations would be equal to $F_h \times F_w \times C$ and for pooling, the number of additions would be equal to I_c [26].

$$I_r = (\frac{(W - F_h + 2P_h)}{S_h} + 1) \times (\frac{(H - F_w + 2 \times P_w)}{S_w} + 1) \tag{3.1}$$

$$I_c = F_h \times F_w \times C \tag{3.2}$$

3.2 CNN computational analysis

As stated before regular CNN's are stacked convolution blocks and exhibit same memory and computational requirements for hierarchical blocks, computational complexity reduces from input to output as after each convolution block input gets reduced

to lower dimension and so on until it matches up to labels.

SqueezeNet [27] architecture was shown in Fig-2.2 and Table-5.8. Most compute and memory intensive is first convolution layer because it operates on the input of size 224x224x3 and as after each layer number of filters increase progressively from 96 filters to 1000 filters but memory required reduces as multiple filters apply on the same input.

Fig-3.3 shows the relative execution time comparison of layers in SqueezeNet and Fig-3.2 shows the comparison of input memory for each layer between GEMM and DC. As shown in Fig-3.2 and Fig-3.3 first convolution layer, conv0 is most compute intensive and has most memory requirement and usually is same for any CNN and optimizing this layer would be advantageous.

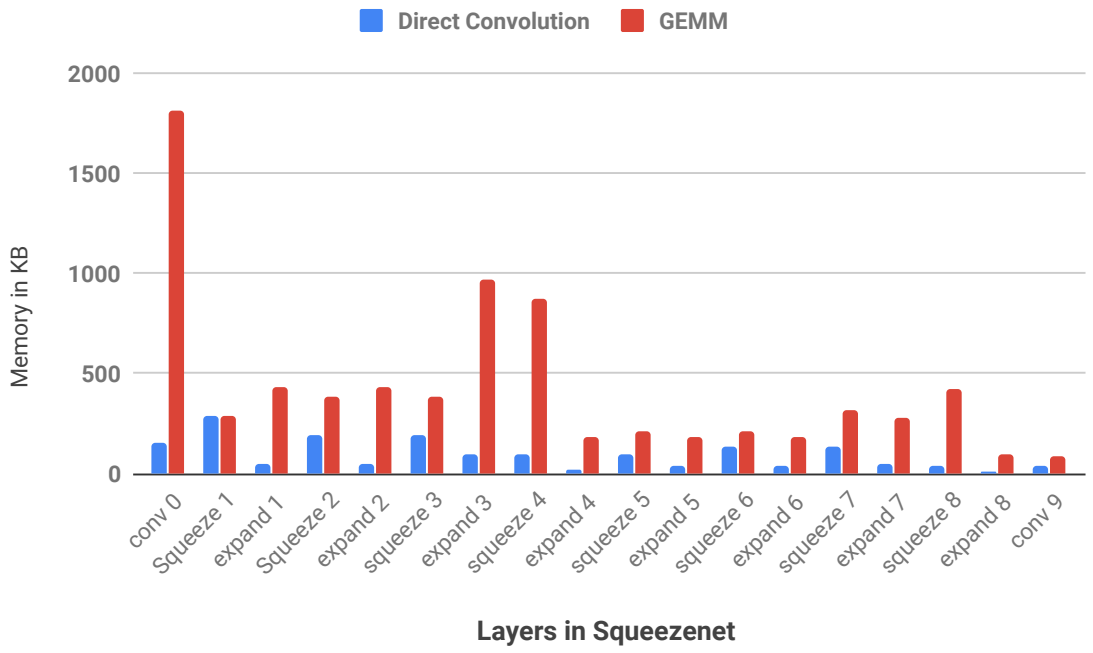


Figure 3.2: Memory requirement comparision between GEMM and Direct Convolution

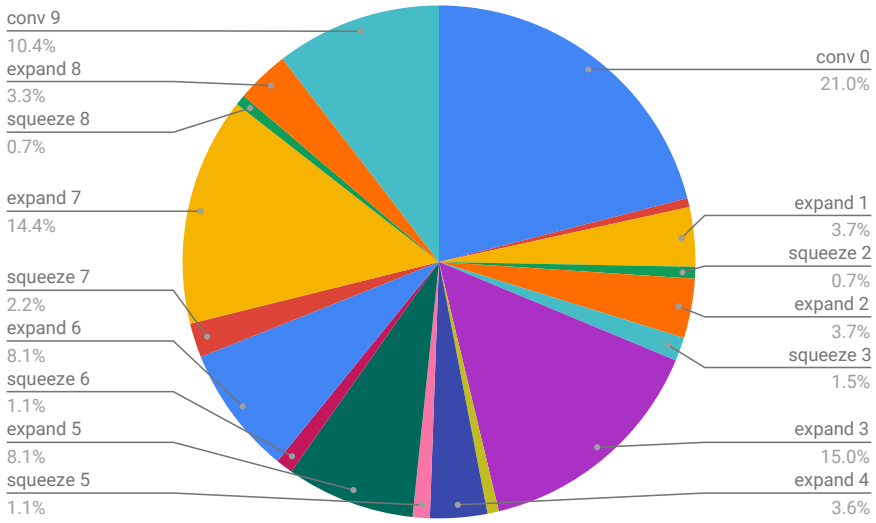


Figure 3.3: Relative Execution time of SqueezeNet layers

As shown in Fig-3.2 and Fig-3.3 first convolution layer, conv0 is most compute intensive and has most memory requirement compared to other layers as first convolution layer operates on the input image and has biggest convolution filter size as shown in eq 3.2 filter size is proportional to the im2col data duplication. [3].

Because of im2col data duplication, input image size multiplies times based on the size of convolution filter and size, For example, in the case of SqueezeNet with 7×7 filters with stride 2, im2col duplicates approximate seven times. As GPU's are coming with large memory, most deep learning libraries accept the trade-off for the speed up and use the GEMM Acceleration. Fig-3.3 shows the memory requirement comparison between GEMM and Direct convolution for SqueezeNet [1].

GEMM proves favorable for GPUs because of the availability of large memory and GEMM solves data dependency between convolution windows and create data independent threads by data duplication maximizing utilization. But in the case of custom hardware with custom datapaths, using direct convolution is much favorable as it requires low memory buffer and data paths can be designed to match the convolution computation. [12].

3.2.1 Capsnet computation analysis

As discussed above, in Capsnet with capsule activations or vector activations with pose and presence information, primary capsule layer generates multiple patterns of lower level features with convolution layer, and dynamic routing algorithm is used to route the lower level features to higher level features. As shown in Table-3.1, dynamic routing algorithm's computational reflection as weight matrix translated into multiple matrix multiplications in step 4 and step 6 in the table. Thus for each routing iteration, two matrix multiplications on the input of size $1152 \times 1 \times 8$ multiplied weights to produce 16×10, in the case of MNIST Capsnet. MNIST capsnet has an input size of $28 \times 28 \times 1$, In case of input with larger size the computation gets much complex.

Table 3.1: Computation translation of dynamic routing algorithm in to matrix operations

S No	Computational Translation of algorithm
step 0	u is output of previous layer; v is output of current layer
step 1	b is tensor or coefficients of routing between u and v Initialized to zero
step 2	r being number of routings; for r iterations:
step 3	b = softmax(b)
step 4	v' = u.b (dot product or Matrix Multiplication)
step 5	v = Squash(v')
step 6	b = b + u.v (dot product or Matrix Multiplication)

Fig-3.4 shows the computational and memory breakdown of Capsnet architecture, Capsulenet computational Analysis, In Capsnet architecture, only first convolution layer and primary capsule layer has a convolution layer involved in this. Fig-3.4 reports the input size difference in GEMM [3] and direct convolution in first convolution and primary capsule layer. Due to large convolution filter size and number in capsnet

GEMM causes 50 times data redundancy in the first convolution layer because of its inputs size and nearly seven times in convolution layer in primary capslayer.

Fig-3.5 shows relative execution time of Capsnet execution time layer-wise. As shown, even though Capsnet architecture has two 256, 9×9 convolution layers [20], the majority of computation time is in dynamic routing layer as it contains iterative large matrix multiplications two times per iteration is same as primary capsule layer. Primary capsule layer and dynamic routing take similar time for 3 number of routings and increase in the number of routing would linearly scale the execution time.

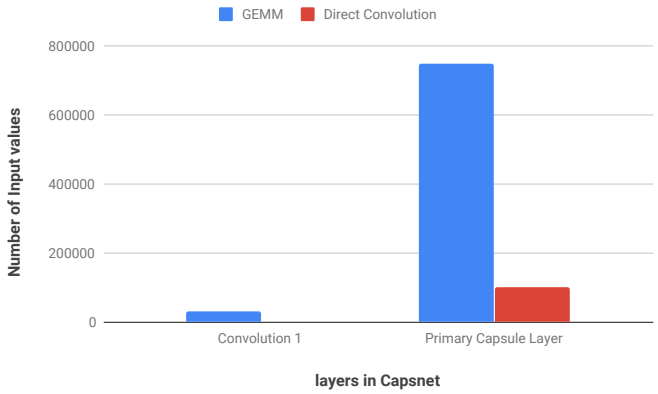


Figure 3.4: GEMM vs Direct convolution input size comparision for MNIST input

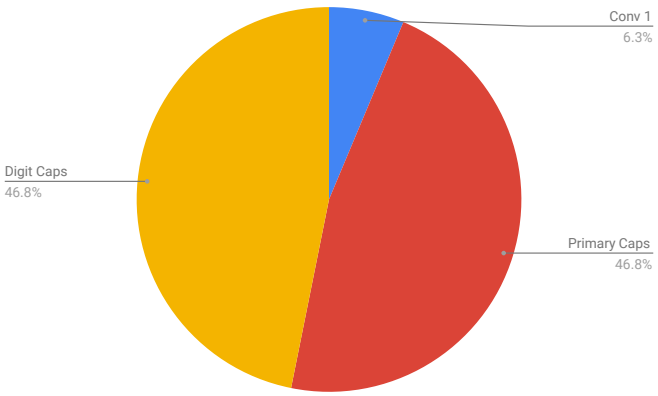


Figure 3.5: Relative execution time of Capsnet layers for MNIST input

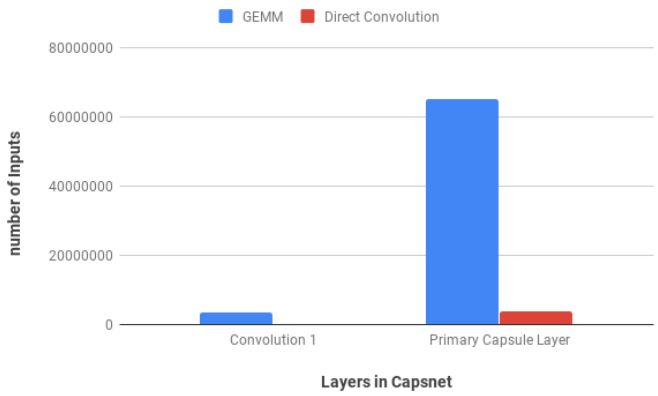


Figure 3.6: GEMM vs Direct Convolution Input Memory Size in Capsnet for a Input of 128 x 128

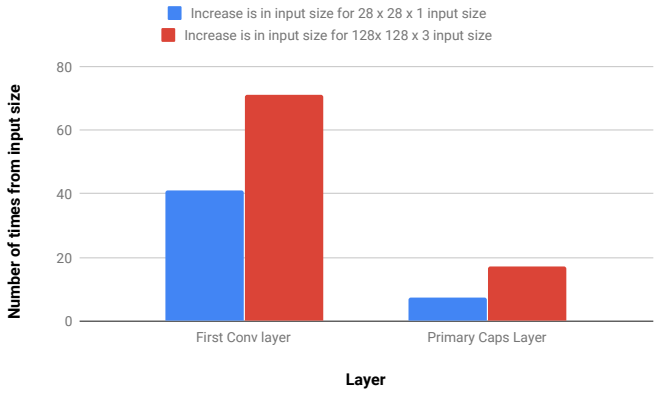


Figure 3.7: Number of times the input size increase due to GEMM Call for multiple inputs sizes in Capsnet

Fig-3.6, shows the input memory size comparision between GEMM vs Direct convolution for Capsnet architecture on a input of size 128 $\times$128 $\times$ 3. As it can be seen in first convolution layer input size increased from 49152 values to 3499200 values increasing input size by 71 times and in primary capsule layer input size from 3686400 values to 65028096 values increasing nearly 17 times. Fig-3.7 shows the increase in size due to GEMM im2col transformation in capsnet for 28 $\times$ 28 $\times$ 1 input size and 128 $\times$ 128 $\times$ 3 size. As shown as input size increases data dupication due im2col increases memory requirement.

CHAPTER 4: APPROACH

This section discusses the algorithmic implementation of 1D Convolution replacement layer and then about 1D Accelerator design.

4.1 Proposed 1D Convolutional Replacement Training

In accelerator design, a considerable amount of effort being done to reduce memory access. This approach utilizes streaming near sensor design to ensure the FM data does not touch main memory and therefore the only memory accesses needed are for kernel data. Avoiding streaming data on on-chip memory also eliminates the memory access of this data, however as new CNN topologies emerge, this option will not be scalable. The solution for this is the reduction in the total amount of parameters. To achieve this by altering the first layer of convolution (which is the most compute-intensive layer) to utilize 1D convolution kernels at the training stage. With this, backward propagation will minimize the gradients of 1D convolutions both at Y and X dimensions and use the train 1D Convolution replacement CNN for inference.

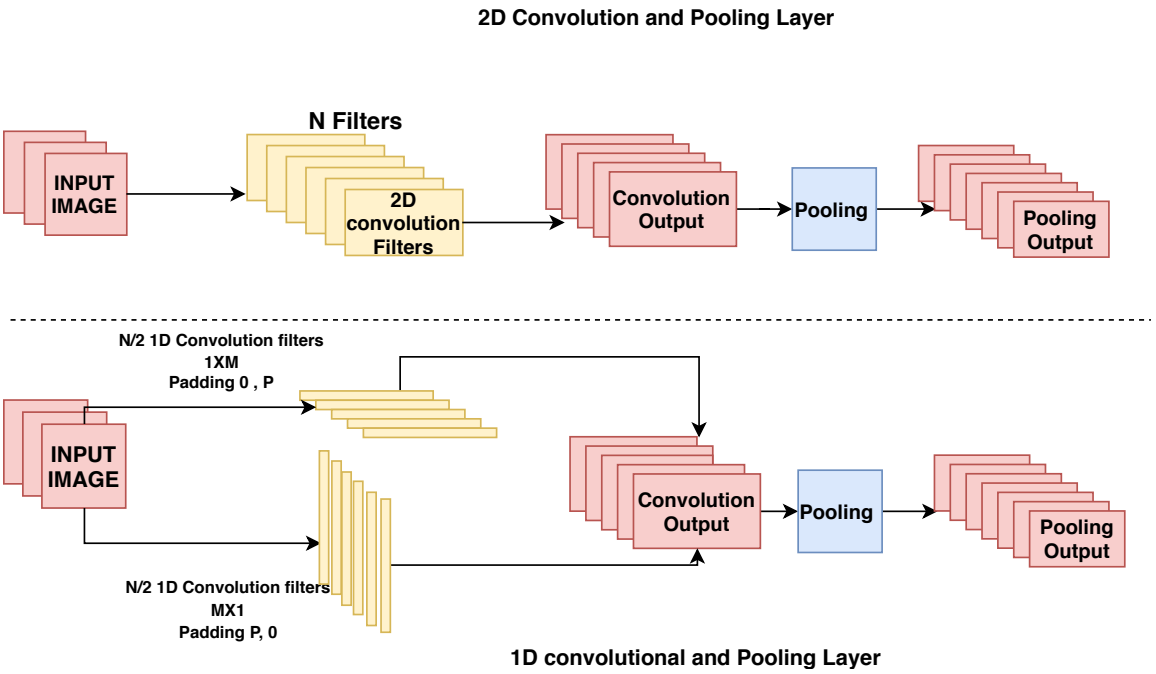


Figure 4.1: Implementation of 1D CNN replacement layer

Fig-4.1 presents proposed 1D convolution optimization. In 1D Convolution replacement layer separate filters for X and Y dimensions and then combining the results by concatenating the feature maps produced by vertical and horizontal convolutions. As a result, there are M/2 parallel $N \times 1$ and $1 \times N$ filters each pair corresponding to a single 2D Convolution layer $N \times N$ filter. As filters operate in parallel on the input image, produce different output for $N \times 1$ and $1 \times N$ and are not concatenable. To make them to same size Input images are padded on both height (H P_h, H P_w), and width (W P_h, W P_w) to produce the same size of output for each dimensional filter. For an Input Image of Size W, H, D is the width, Height, and Depth of image as filters are transposes of each other width-wise stride of a filter would be equal to the height-wise stride of another dimensional filter.

$$H \times P_w = W \times P_h = P \tag{4.1}$$

$$H \times P_h = W \times P_w = \frac{N-1}{2}, where\ N\ is\ filter\ size \tag{4.2}$$

4.1.1 1D convolution output Size increase

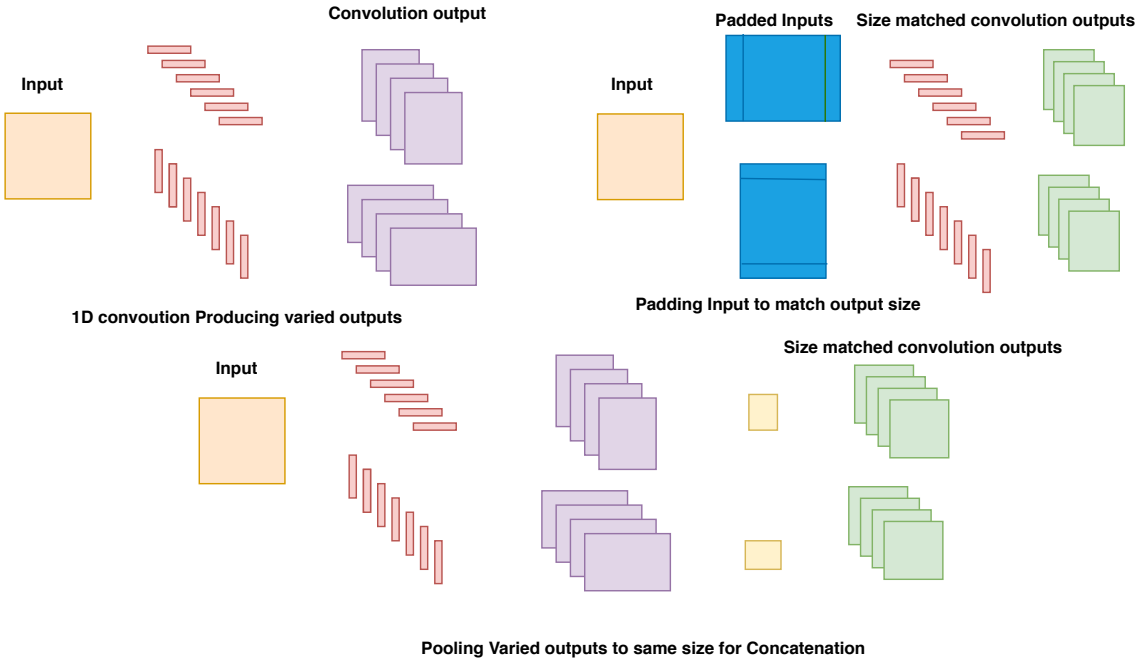


Figure 4.2: 1D Convolution output size matching

As shown in Fig-4.2, because of 1-Dimensional convolution application, the Convolution output size is not same in both width and height dimensions, and also output size of convolution with respect to primary dimension filter will be smaller than that dimension output of the non-primary-dimension filter. This difference is equal to the difference of both dimensions of filter divided by the stride of the convolution layer. Input Image of Size $W \times H \times C$ convolution filters of size $F \times F$, input zero padding of P_h and P_w Stride of S as shown in eq 4.3, increase in output size after 1D-convolution O_{1D}. Hence in case of convolution with stride 1, 1D convolution produces an output of size same as inputs.

In any CNN number of convolution weights don't change with a change in output size of the first convolution, But in case of a network with fully connected layer number of inputs change increasing the number of connections and increasing the number of weights. Increase in output size due to 1D convolution replacement can

be avoided in the following ways:

$$O_{1D} = \frac{F-1}{S} \tag{4.3}$$

- Replacing fully connected with 1×1 convolution and average pooling layer as in SqueezeNet.
- In most of the CNNs, pooling layer is followed by convolution layer, by applying the different pooling filters on $N \times 1$ and $1 \times N$ filters in 1D convolution replacement layer.
- Increasing the pooling filter of the size of pooling layer before fully connected layer.

Overall, one dimensional-training and inference can lead to a significant (exponential) theoretical reduction in memory and computation demand of CNN layer. In the case of SqueezeNet, 48 sets of 7×1 and 48 sets of 1×7 filters replace the 96 convolution 7×7 filters, which finally output 96 concatenated feature maps. Thus, the replacement layer for the first Convolution in SqueezeNet reduces the number of parameters in the first layer by seven times [1].

4.1.2 Capsnet 1D replacement

In the case of Capsnet [20], regular 1D replacement would increase the output size increasing the number of parameters in further layers. This is avoided using a new approach ,Fig-4.3 shows explains the approach. In this input trimming approach, we stripped the edge pixels which do not carry important feature information, i.e, in case of N x 1 filter, horizontal features are captured, and vertical boundary pixels will be insignificant, and output size increase is due to convolution filter dimension one operation on vertical dimension. Hence ignoring vertical boundary pixels in convolution computation with horizontal filter and ignoring horizontal boundary pixels with convolution computation with vertical filters will not affect the feature space at

large. Using this approach Capsnet 1D was trained on both MNIST and CIFAR10, and there was no drop in accuracy in case of MNIST, but in case of CIFAR10 accuracy dropped by 4% compared regular 1D trained Capsnet and 0% compared to original 1D Capsnet. This drop in accuracy was minimum in case of MNIST because MNSIT dataset has black background with no noise, and boundary pixels ignored are mostly black and in case of cifar10 drop in accuracy compared to original case is zero, because most images are image centered in both training and validation dataset [8],[21] and boundary pixels are usually noise or background mostly. Larger the input size is less significant the boundary pixels become.

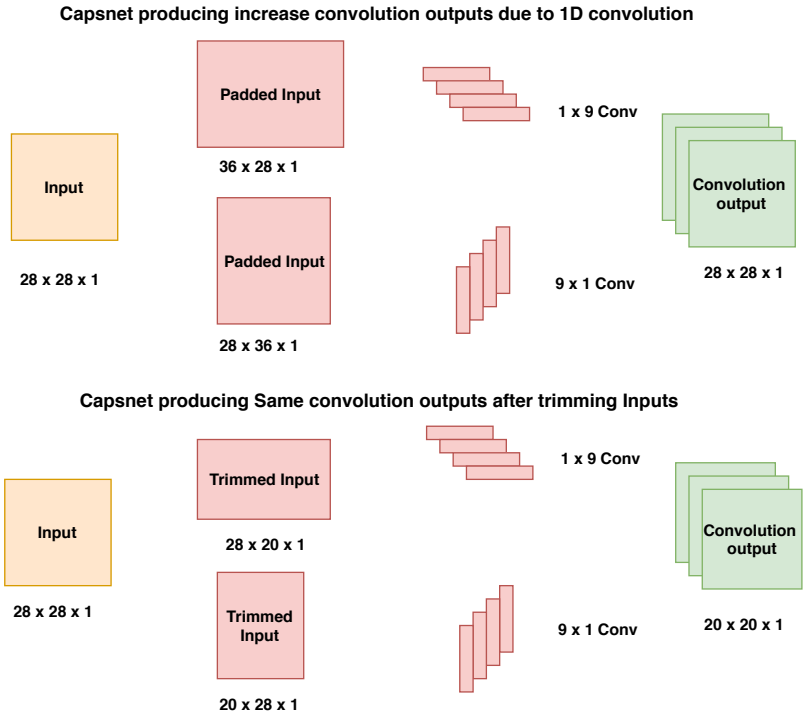


Figure 4.3: 1D Capsnet input trimming

For ease of implementation, instead of stripping the input differently for each dimensional convolution in 1D replacement training boundary pixels belonging to the convolution output due to boundary pixels were stripped or trimmed. By stripping the convolution output to the same size as 2D convolution output, number of parameters did not increase.

4.1.3 1D Convolution in GPUs

Even though with 1D Convolution replacement number of parameters in the first layer reduces by N times for a N x N Convolution filter. On GPUs, all Deep Learning libraries call each kernel sequentially one after the other on same input differently padded for N x 1 Convolution and 1 X N Convolution. Increasing the overall execution time. But in case of custom hardware accelerator designs, 1D convolution replacement would be advantageous. In the case of the custom accelerator, custom datapaths are synthesized to meet the computation requirements and designing custom accelerator paths for 1D design. The number of values required to compute a convolution output reduced N times. This 1D design would require less buffer size to hold computation, and N multiplications to compute one convolution output.

4.2 1D Architecture

This section discusses the architecture design that was used to exploit the advantages of the 1D replacement approach [1]. On Algorithm aware architecture design 1D convolution replacement approach is implemented in [1] was configured to utilize the natural parallelism of CNNs. The architecture can be reconfigured to map to any network topology allowing architecture to handle the first layer high streaming data size and intensive computation, efficiently.

The proposed 1D hardware accelerator in [1] is designed to do inference analytics at the edge has three main parts: (1) Convolutional Processing Element (CPE) to perform convolution on weights and image pixels. (2) Aggregation Processing Element (APE) to sum outputs of the convolution of Red/Green/Blue channels and convert negative values to zeros. (3) Pooling Processing Element (PPE) that performs max-pooling on the output of APE. In this subsection, we will discuss these parts in a 1D convolution hardware accelerator.

4.2.1 CPE

From the architecture design perspective, 1D convolution affects the design of buffers and compute engine or design which is called the CPE.

Since horizontal and vertical convolutions are performed on the same input image but totally independent from each other, the proposed architecture has two components that run in parallel. (1) A horizontal component that is in charge of re-arranging image pixels for horizontal convolution and passing them through MAC units. (2) A vertical component that is in charge of re-arranging image pixels for vertical convolution (which is a different order than that of horizontal convolution) and passing them through MAC units. These components are discussed as follow:

- Horizontal Convolution: Convolution windows are horizontal which means there are no two pixels from different rows in the same window. This results in storing only one single row at a time with a 1D-line buffer. This sub-module consists of one single row of Random Access Memory, as large as the image row. Due to the stride in convolution, Convolution is performed on every other row. The pixels are stored in 1D line buffer as they come from the camera. As the first row read finishes, further reads from the 1D-line buffer continue. MACs are often re-reading the same pixel data as multiple convolution windows share pixel data between two windows because of convolution stride. By the time MACs finish reading the row 0 from RAM, row 2 of the streaming image starts to arrive, and since MACs have completed operations on row 0, RAM is used to store the row 2 since this row is part of the horizontal convolution. Lower block in Fig-4.4 depicts the way that the 1D-line buffer reads in every clock cycle and illustrates the sub-component in charge of doing horizontal convolution.

- Vertical Convolution: Convolution windows are vertical, which means pixels from the same column but seven different rows. In this case, the buffer needs to store the first seven rows of the image before we start the convolution. However, because of stride Convolution is done only on even columns and therefore, the

only store even columns need to be stored. Thus, the smallest amount of data needed to store for doing the vertical convolution is 7×114 (ceil(227/2)=114) pixels. The 2D-line buffer consists of 9 rows of FIFOs with 114 elements each (i.e., pixels, i.e., bytes). These FIFOs are independent of each other and re-readable. Re-readable FIFOs are rows of memory elements that still keep the data after it is read but have writing and reading data that is done sequentially. The first seven rows store the data that is being worked on, and when they are full, we start reading from the 2D-line buffer and performing the convolution. It takes 7*114 clock cycles to read all the data in 2D-line buffer once. As we are convolving the data from the first seven rows, streaming input data is stored in 2 remaining rows. Writing data into 2D-line buffer happens much slower than reading data from it due to the desired frame rate. This means those two extra lines of FIFO will not be full until after convolution window is slid all the way through the first seven rows and all the convolutions are performed. Once done with one convolution window, we shift the row indexes and reuse the rows with old pixels to store the input of the new ones. Upper block in Fig-4.4 depicts the way that the 2D-line buffer is read in every clock cycle and illustrates the component in charge of vertical convolution.

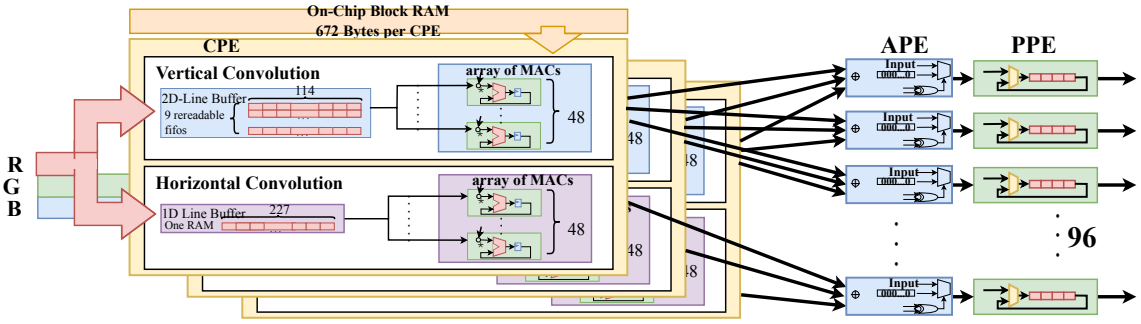


Figure 4.4: complete 1D replacement layer hardware architecture

4.2.2 Full layer architecture

Implemented the first layer of a neural network for image processing applications with 1D architecture [1], there should be one CPE per each input channel, i.e. 3 CPEs

in total. According to Squeeznet topology, there are 96 kernels in the first layer, which means each CPE should have 96 MACs meaning 96 outputs (48 for horizontal 48 for vertical convolution) to implement kernel parallelism (where all the kernels are used at the same time). These each CPEs outputs are added together in APEs. There should be one APE per kernel which makes it 96 APEs total. Outputs of these 96 APEs go to 96 independent PPEs, which are located right after APEs. Fig-4.4 shows a detailed illustration of the proposed architecture for the first layer of SqueezeNet topology.

To conclude, [1] 1D architecture overview with the understanding that our algorithm optimization translates to the benefits of 7 times less kernel memory (2016 B). The total memory required to store pixels before convolution is 1253 Bytes, reducing the buffer size by almost half, and the operation frequency that drives the computation of our architecture is seven times smaller due to the reduced number of operations per convolution.

CHAPTER 5: RESULTS

In this section, we compare the architectural results of 1D convolution replacement with their original non-replacement counterparts and architectural evaluation of 1D convolution effect on accelerator design.

5.1 Experimental Setup

1D replacement explained in approach is tested on various datasets MNIST [8], CIFAR10 [21], CIFAR100 [21] and Imagenet [16] on various networks like Resnet 18, Resnet 34, Resnet 50 [9], Capsnet [20], ZFNET [19], Squeeznet [27], and Alexnet [10]. Comparison of training and validation accuracies and training and validation losses are reported, for experimentation official datasets were used without any change.Tables 5.2, 5.8, 5.9, 5.10, 5.7, 5.6, 5.5, 5.4 are the network architectures used in experimentation and Table 5.1 reports the training hyper-parameters used in training the networks. Experiments on MNIST, CIFAR10, and CIFAR100 were trained using Keras [28] library and experiments on Imagenet are trained using Caffe library. All the input data for all the datasets are min-max normalized. MNIST experiments are performed with input size of $28 \times 28 \times 1$, CIFAR10 and CIFAR100 experiments are performed with input size of $32 \times 32 \times 3$, and Imagenet inputs were centre cropped to $224 \times 224 \times 3$. [24].

Table 5.1: Training Hyper parameter used for training networks

Experiment	Training Hyper-Parameters
MNIST CNN	Adadelta Optimizer, Categorical cross entropy loss, Learning rate: 1.0
MNIST and CIFAR10 Capsnet	Adam Optimizer, Learning rate: 1.0
CIFAR10 and CIFAR100 Resnet-18 and Resnet-50	Adam Optimizer, Categorical cross entropy loss, Learning rate: 0.001
CIFAR10 and CIFAR100 ZFNET	Adam Optimizer, Categorical cross entropy loss, Learning rate: 0.001
Imagenet Squeezenet	Base Learning rate: 0.04, Polynomial learning policy, power 1.0, 170000 Iterations
Imagenet Alexnet	Base Learning rate: 0.01, step learning policy, gamma 0.1,step 1e5, momentum 0.9, weight decay 5e-4, 450000 iterations
Imagenet Googlenet	Base learning rate: 0.01, step learning policy, step 3.2e5,gamma 0.96, weight decay 2e-4, 1000000 iterations

5.2 Algorithmic Evaluation

This result shows that the replacement would just cost minor loss in accuracy but will provide a low power accelerator favorable design.

Table 5.2: Capsnet and 1D Capsnet Architecture used to train on MNIST and CIFAR10 Datasets

Capsnet	Capsnet 1D V1
conv 9 x 9 / 1 (x256)	Conv 9 x 1 / 1 (x 128) conv 1 x 9 / 1 (x128) Concatenate
dropout	
conv 9 x 9 / 2 (x256) reshape	
Lambda Activation Matrix Multiplication (x Number of Routings) Reshape (x Number of Classes)	

Table 5.3: Capsnet and 1D Capsnet V2 Architecture with Input trimming used to train on MNIST and CIFAR10 Datasets

Capsnet	Capsnet 1D V1
conv 9 x 9 / 1 (x256)	Conv 9 x 1 / 1 (x 128) conv 1 x 9 / 1 (x128) Concatenate
trim input	
dropout	
conv 9 x 9 / 2 (x256) reshape	
Lambda Activation Matrix Multiplication (x Number of Routings) Reshape (x Number of Classes)	

Table 5.4: MNIST CNN and 1D CNN Architecture

mnist CNN	Mnist CNN 1D
conv 3 x 3 (x 32)	conv 3 x 1 (x16) | conv 1 x 3 (x 16) concatenate
conv 3 x 3 (x 64)	conv 3 x 3 (x 64)
maxpool 2 x 2	maxpool 4 x 4
dropout 0.25	dropout 0.25
Fully connected	Fully connected
dropout 0.5	dropout 0.5

Table 5.5: Resnet-18 and 1D Resnet-18 Architecture

Resnet 18	
18 Layer	18 Layer 1D
conv 7 x 7 / 2 (x64)	conv 7 x 1 / 2 (x32) Conv 1 x 7 /2 (x32) concatenate
maxpool 3 x 3 / 2	
conv 3 x 3 / 2 (x 64) x 2	
conv 3 x 3 / 2 (x 128) x 2	
conv 3 x 3 / 2 (x 256) x 2	
conv 3 x 3 / 2(X 512) x 2	
avg pool	
fully connected	

Table 5.6: Resnet-50 and 1D Resnet-50 Architecture

<table>
<tr><td colspan="2">Resnet 50</td></tr>
<tr><td>50 Layer</td><td>50 Layer 1D</td></tr>
<tr><td rowspan="2">conv 7 x 7 / 2 (x64)</td><td>conv 7 x 1 / 2 (x32) conv 1 x 7 / 2 (x32)</td></tr>
<tr><td>Concatenate</td></tr>
<tr><td colspan="2">maxpool 3 x 3 / 2</td></tr>
<tr><td colspan="2">conv 1 x 1 (x 64) conv 3 x 3 (x 64) conv 1 x 1 (x 256)</td></tr>
<tr><td colspan="2">conv 1 x 1 (x 128) conv 3 x 3 (x 128) conv 1 x 1 (x 512)</td></tr>
<tr><td colspan="2">conv 1 x 1 (x 256) conv 3 x 3 (x 256) conv 1 x 1 (x 1024)</td></tr>
<tr><td colspan="2">conv 1 x 1 (x 512) conv 3 x 3 (x 512) conv 1 x 1 (x 2048)</td></tr>
<tr><td colspan="2">avg pool</td></tr>
<tr><td colspan="2">fully connected</td></tr>
</table>

Table 5.7: ZFNET and 1D ZFNET Architecture

ZFNET	1D ZFNET
conv 5 x 5 / 2 (x96)	conv 5 x 1 / 2 (x48) | conv 5 x 1 / 2 (x48) concatenate
maxpool 3 x 3 / 2	maxpool 3 x 3 / 2
batch norm	batch norm
conv 3 x 3 / 2 (x 256)	conv 3 x 3 / 2 (x 256)
maxpool 3 x 3 / 2	maxpool 3 x 3 / 2
batch norm	batch norm
conv 3 x 3 / 2 (x 384)	conv 3 x 3 / 2 (x 384)
conv 3 x 3 / 2 (x 384)	conv 3 x 3 / 2 (x 384)
conv 3 x 3 / 2 (x 256)	conv 3 x 3 / 2 (x 256)
fully connected 4096	fully connected 4096
dropout 0.5	dropout 0.5
fully connected 4096	fully connected 4096
dropout 0.5	dropout 0.5

Table 5.8: Squeezenet and 1D Squeezenet Architecture

1D Squeezenet		SqueezeNet	
conv 7 x 1/ 2 (x96)	conv 1 x 7/ 2 (x96)	conv 7 x 7/ 2 (x96)	
maxpool 3 x 3 / 2		maxpool 3 x 3 / 2	
fire 2	x16 Sqz, x64 exp1, x64 exp3	fire 2	x16 Sqz, x64 exp1, x64 exp3
fire 3	x16 Sqz, x64 exp1, x64 exp3	fire 3	x16 Sqz, x64 exp1, x64 exp3
fire 4	x32 Sqz, x128 exp1, x128 exp3	fire 4	x32 Sqz, x128 exp1, x128 exp3
maxpool 3 x 3 / 2		maxpool 3 x 3 / 2	
fire 5	x32 Sqz, x128 exp1, x128 exp3	fire 5	x32 Sqz, x128 exp1, x128 exp3
fire 6	x48 Sqz, x192 exp1, x192 exp3	fire 6	x48 Sqz, x192 exp1, x192 exp3
fire 7	x48 Sqz, x192 exp1, x192 exp3	fire 7	x48 Sqz, x192 exp1, x192 exp3
fire 8	x64Sqz, x256exp1, x256 exp3	fire 8	x64Sqz, x256exp1, x256 exp3
maxpool 3 x 3 / 2		maxpool 3 x 3 / 2	
fire 9	x64Sqz, x256 exp1, x256 exp3	fire 9	x64Sqz, x256 exp1, x256 exp3
conv 1 x 1 / 1 (x1000)		conv 1 x 1 / 1 (x1000)	
avg Pool 13 x 13 /1		avg Pool 13 x 13 /1	
Sqz Squeeze 1 x 1: exp1 Expand 1 x 1:: exp3 Expand 3 x 3			

Table 5.9: Alexnet and 1D Alexnet V1 Architecture

Alexnet	1D Alexnet V1	
conv 9 x 9 /4 (x96)	conv 9 x 1 / 4 (x96)	conv 9 x 9 /4 (x96)
max pool 3 x 3 /2	Maxpool 3 x 3 /2 Concatenate	
Batch Norm	Batch Norm	
conv 5 x 5 /1 (x256)	conv 5 x 5 /1 (x256)	
max pool 3 x 3 /2	max pool 3 x 3 /2	
Batch Norm	Batch Norm	
conv 3 x 3 / 1 (x384)	conv 3 x 3 / 1 (x384)	
conv 3 x 3 / 1 (x384)	conv 3 x 3 / 1 (x384)	
conv 3 x 3 / 1 (x256)	conv 3 x 3 / 1 (x256)	
max pool 3 x 3 /2	max pool 3 x 3 /2	
fully connected 4096	Fully Connected 4096	
fully connected 4096	Fully Connected 4096	
fully connected 1000	Fully Connected 1000	

Table 5.10: Alexnet and 1D Alexnet V2 Architecture

Alexnet	1D Alexnet V2	
conv 9 x 9 /4 (x96)	conv 9 x 1 / 4 (x96)	conv 9 x 9 /4 (x96)
max pool 3 x 3 /2	maxpool 3 x 5 /2	maxpool 5 x 3 /2
	Concatenate	
Batch Norm	Batch Norm	
conv 5 x 5 /1 (x256)	conv 5 x 5 /1 (x256)	
max pool 3 x 3 /2	max pool 3 x 3 /2	
Batch Norm	Batch Norm	
conv 3 x 3 / 1 (x384)	conv 3 x 3 / 1 (x384)	
conv 3 x 3 / 1 (x384)	conv 3 x 3 / 1 (x384)	
conv 3 x 3 / 1 (x256)	conv 3 x 3 / 1 (x256)	
max pool 3 x 3 /2	max pool 3 x 3 /2	
fully connected 4096	Fully Connected 4096	
fully connected 4096	Fully Connected 4096	
fully connected 1000	Fully Connected 1000	

Table 5.11: Googlenet and 1D Googlenet architecture

	1D Googlenet	Googlenet
Layer Type	Filter	Filter
convolution 1	7 x 1 / 2 (x48) 1 x 7 / 2 (x 48)	7x7/2 (x64)
Concatenate	concat	
max pool	3x3/2	
convolution 2	3x3/ 1 (x192)	
maxpool	3x3/2	
inception (3a)	(x256)	
inception (3b)	(x480)	
maxpool	3x 3/2	
inception (4a)	(x512)	
inception (4b)	(x512)	
inception (4c)	(x512)	
inception (4d)	(x528)	
inception (4e)	(x832)	
maxpool	3 x 3/ 2	
inception (Sn)	(x832)	
inception (Sb)	(x1024)	
avgpool	7x7/ 1	
dropout	0.4	
linear	(x1000)	
Softmax		

5.2.1 MNIST Results

In this section, as shown in Fig-5.1 and Fig-5.2 training and validation accuracy and training and validation loss of simple 5 layer CNN, mnistCNN with architecture as shown in Table-5.4 and its 1D replacement counterpart and Capsnet with architecture

as shown in Table-5.2 and its 1D replacement counterparts 1D Capsnet V1 as shown in Table-5.2 and also as 1D Capsnet V2 with input trimming to avoid increase in output size as shown in Table-5.3 were compared. In the case of MNIST CNN, there was a drop of 0.3% validation accuracy. In case of Capsnet, with network architecture Table-5.2 original Capsnet has a validation accuracy of 98.87% and 1D replacement Capsnet with network architecture Table-5.2, 1D Capsnet V1 has a validation accuracy of 99.4%, increase in accuracy is because of increase in network parameters in routing layer due increase in size of convolution output. In the case of 1D Capsnet V2 with network architecture Table-5.3 validation accuracy is 99.34% with no drop in accuracy. There was no drop in accuracy because MNIST Dataset had figures centered and usually noise or background exists in boundaries and MNIST has black backgrounds.

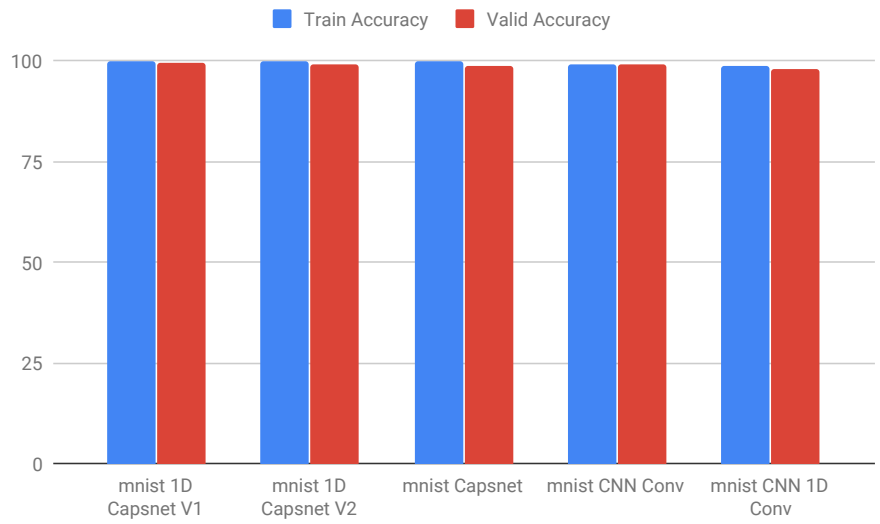


Figure 5.1: Train and Valid accuracy comparision of various networks against 1D replaced versions trained on MNIST dataset

Figure 5.2: Train and Valid Loss comparison of various networks against 1D replaced versions trained on MNIST dataset

5.2.2 CIFAR-10 Results

In this section, as shown in Fig-5.3 and Fig-5.4 compares the training and validation accuracy and training and validation loss of Resnet-18 with architecture as shown in Table-5.5, ZFNET with architecture as shown in Table-5.7 and Capsnet, 1D Capsnet V1, and 1D Capsnet V2 with architectures as shown in Table-5.2 and Table-5.3.In the case of Capsnet, there was no drop in accuracy between Capsnet and 1D Capsnet V1 and 1D Capsnet V2. There was no drop in accuracy in 1D Capsnet V2 with Input trimmed version as CIFAR10 also had images centers and boundary pixels are mostly noise. In the case of ZFNET, Original ZFNET has a validation accuracy of 74.95%, and 1D ZFNET had a validation accuracy of 71.48% with a drop in accuracy of 3.47%. In the case of Resnet-18, original Resnet-18 had a validation accuracy of 85.7%, and 1D Resnet-18 has a validation accuracy of 83.1% with a drop in accuracy of 2.6%.

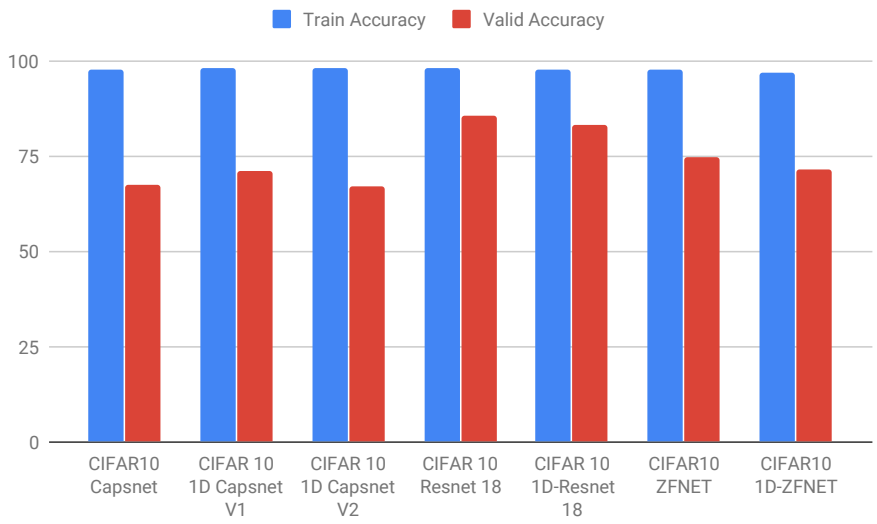


Figure 5.3: Train and Valid accuracy comparision of various networks against 1D replaced versions trained on CIFAR 10 dataset

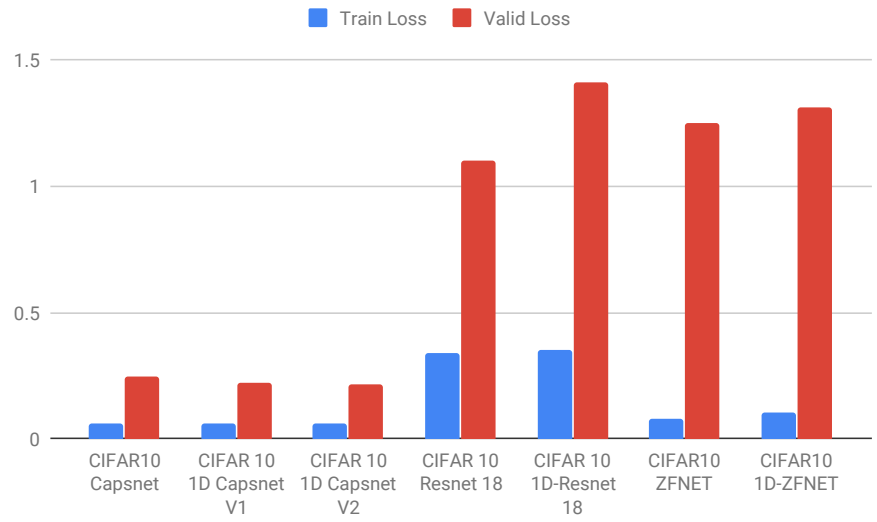


Figure 5.4: Train and Valid Loss comparision of various networks against 1D replaced versions trained on CIFAR 10 dataset

5.2.3 CIFAR-100 Results

In this section as shown in Fig-5.5 and Fig-5.6 compares the training and validation accuracy and training and validation loss of Resnet-50 with architecture as shown in Table-5.6, ZFNET with architecture as shown in Table-5.7. In case of ZFNET,

Original ZFNET has a validation accuracy of 58.81% and 1D ZFNET had a validation accuracy of 57.948% with a drop in accuracy of 0.86%. In case of Resnet-50, original Resnet-50 had a validation accuracy of 57.94% and 1D Resnet-18 has a validation accuracy of 56.35% with drop in accuracy of 1.59%.

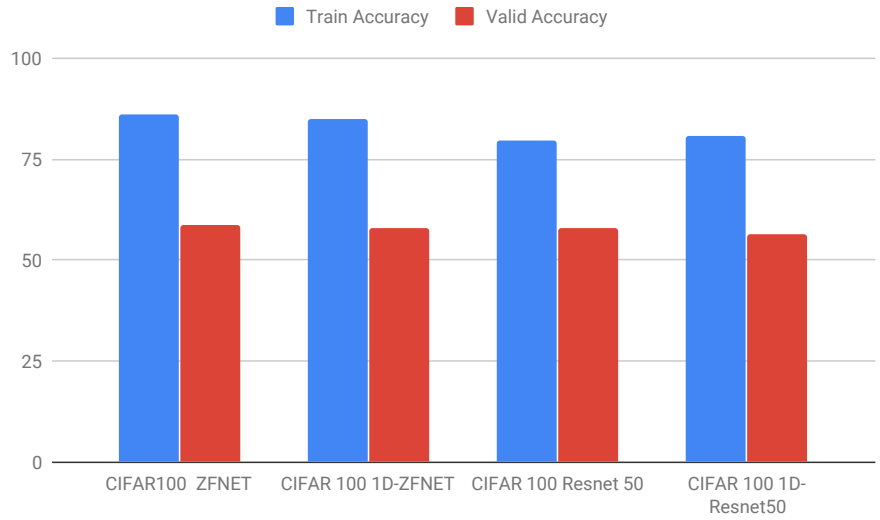


Figure 5.5: Train and Valid accuracy comparision of various networks against 1D replaced versions trained on CIFAR 100 dataset

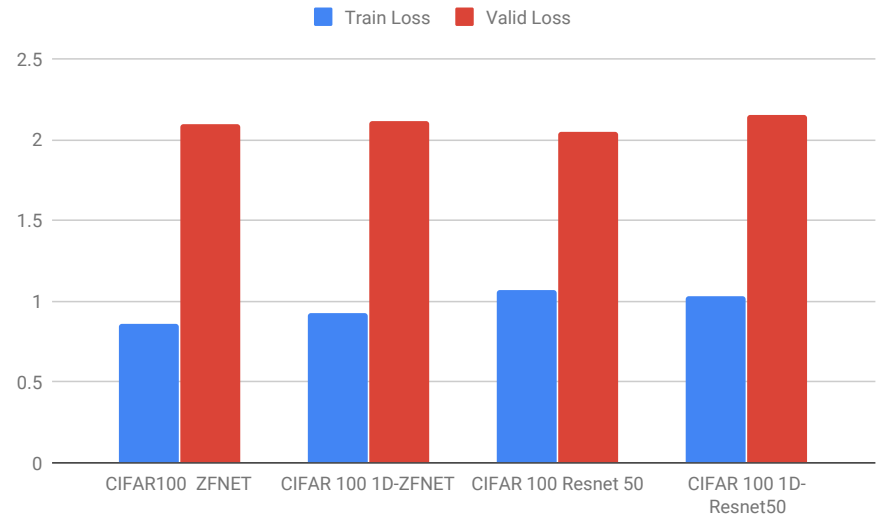


Figure 5.6: Train and Valid loss comparison of various networks against 1D replaced versions trained on CIFAR 100 dataset

5.2.4 Imagenet results

Fig-5.7 shows the train and validation accuracy comparison of various networks architectures on Imagenet dataset. Networks used are SqueezeNet as shown in Fig-5.8, Alexnet as shown in Table-5.9,Table-5.10 and Googlenet as shown in Table-5.11 compared against 1D replacement layer applied in respective networks on the first layer. As shown in Fig-5.7 drop in accuracy is minimum due to 1D replacement. In the case of Alexnet and SqueezeNet 1D replacement layer accuracies are compared against BVLC official accuracies [29]. In case of Googlenet accuracies are compared again bvlc googlenet accuracy [29] and accuracy achieved training Googlenet and 1D Googlenet with prescribed training hyperparameters [18], trained from scratch.

In Alexnet [10] due to the presence of FC layer after final convolution, increment in convolution output size effects the number of parameters in FC layer. In Alexnet after 1D replacement, 1D Alexnet V1 output of last convolution layer Conv5 becomes $256\times7\times7$ while in case of regular Alexnet output size of convolution size is $256\times6\times6$, Even though change in size just 256, FC layer associates with 1000 labels making the FC layer weights increase by 256×1000. To avoid this as explained in CNN analysis section, Rectangular pooling of filter sizes 5×3 and 3×5 are applied on 1×7 and 7×1 convolution outputs after 1D replacement layer making output of convolution size equal to regular Alexnet case and avoid parameter redundancy, In fig 5.7 1D Alexnet V1 represent the 1D replacement network without per dimensional pooling and 1D Alexnet V2 represents the 1D Alexnet network with per dimensional rectangular pooling. As is it can be seen in Fig-5.7 that this transformation of convolution outputs with rectangular pooling did not have any effect on accuracy.

In case of Googlenet [18], while replacing 64, 7×7 convolutions in the first layer with 32, 7×1 and 32, 1×7 dimensional filters, Network large accuracy drop in this case, But after replacing first convolution layer with 48, 7×1 and 48, 1×7 filters network converged to accuracy with less drop in accuracy compared BVLC model at

same iterations. In Fig-5.7 plot represented by googlenet* reports the accuracy after the number of iterations prescribed by [29], 1D googlenet* is the accuracy of the 1D replacement Googlenet with the same number of iterations and BVLC-Googlenet [29] is accuracy reported in bvlc caffe official repository. Even in SqueezeNet and Alexnet 1D Convolution replacement training network didn't converge to bvlc reported accuracies even after the prescribed number of iterations for training. SqueezeNet needed to be trained three times the specified number of iterations for convergence [27],[10],[18] to closest to reported SqueezeNet accuracy. Drop in accuracy in the case of Googlenet with 1D replacement with 32, 7×1 and 32, 1×7 was because of inability of 64, 1D dimensional feature to represent the 64, 2D features. But 96, 1D features were able to represent the 64, 2D features and drop in accuracy after replacing 64, 7×7 filters with 48, 7×1 and 48, 1×7 filters is just 2.6% compared to original googlenet trained from scratch by me for prescribed number of iterations by bvlc. Due to the increase in the number of channels in convolution output after 1D replacement in 1D Googlenet, the number of parameters in further layers increased by 55286. Thus in case of SqueezeNet drop in accuracy due to 1D Convolution replacement is 0.5%, in case of Alexnet drop in accuracy is 2.3%, and in case of Googlnet drop in accuracy is 2.6% compared to Googlenet trained from scratch with bvlc prescribed hyper-parameters. * represents the Googlenet networks trained from scratch and BVLC googlenet is bvlc reported official Googlenet accuracy.

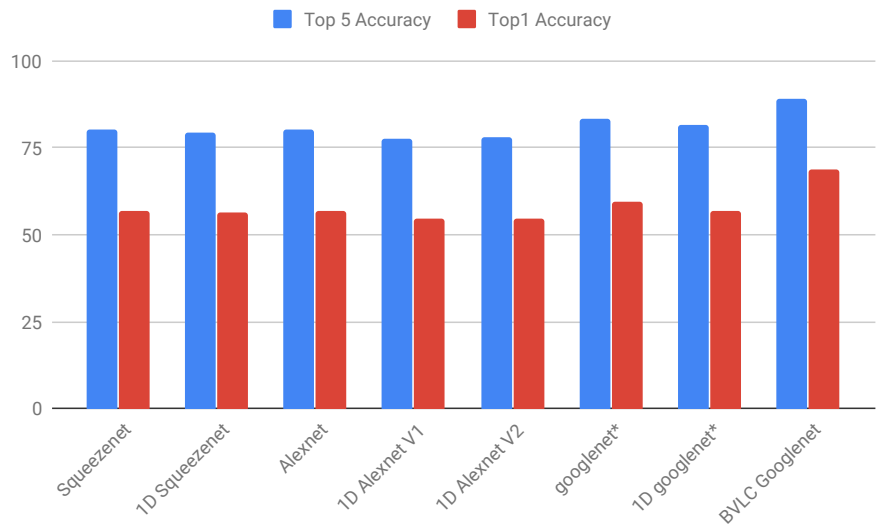


Figure 5.7: Test Top1 and Top5 accuracy comparision of various networks against 1D replaced versions trained on Imagenet dataset

This section is focused on the evaluation of 1D convolution on SqueezeNet and trained our 1D-SqueezeNet using Caffe on Nvidia Tesla P100 [30] with CUDNN [31] acceleration. The Fig-5.8 reports accuracy vs. iteration between original SqueezeNet and 1D-SqueezeNet until 170000 Iterations. 1D version of SqueezeNet has approximately 2% difference in accuracy compared with Original SqueezeNet at 170000 iterations, and after 510000 iterations final top5 accuracy of 79.89% and top-1 accuracy of 56.45% and original SqueezeNet has final top-5 accuracies of 80.59% and top-1 accuracy of 56.7% [29],[27]. Although accelerator design was implemented for SqueezeNet architecture, many networks are tested with 1D replacement to establish the applicability of 1D replacement layer. As shown in Fig-5.1 in results 1D replacement was also tried on Capsnet architecture, even though 1D replacement being unfavorable in Capsnet due to stride one convolution, input trimming approach used help in controlling the increase of parameters. To evaluate the generalizable nature of 1D Convolution replacement, 1D Convolution replacement was trained on Capsnet to prove that combinations of feature maps produced from 1D replacement convolution layer with primary Capsule layer were able to route to higher dimensional features

without any loss in accuracy.

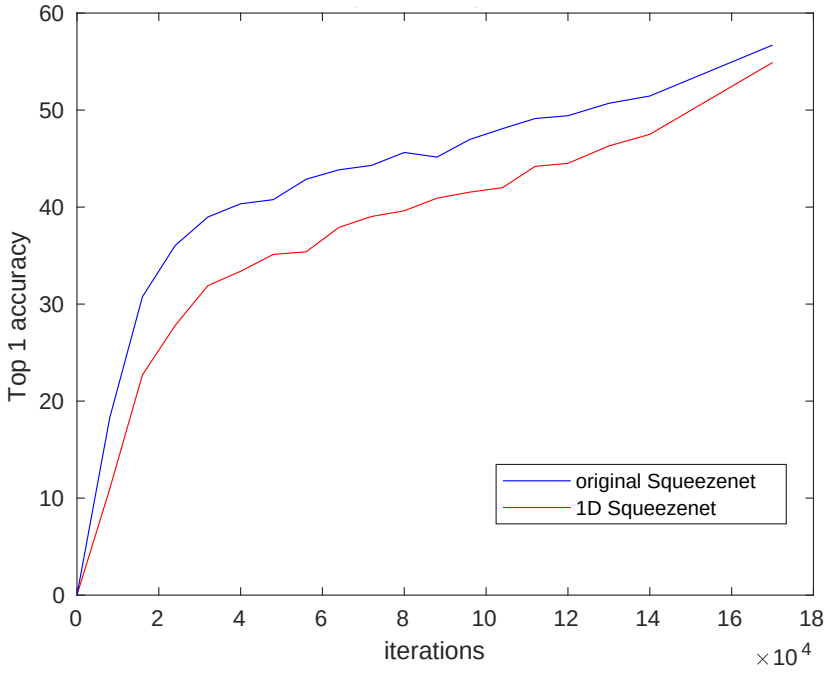


Figure 5.8: SqueezeNet Vs 1D SqueezeNet

Table 5.12 compares specs between SqueezeNet and 1D-SqueezeNet, It can be seen clearly with less than 1% drop accuracy, We could achieve 85.71% reduction in parameters in first convolution layer enabling an edge friendly design for first convolution layer.

Table 5.12: SqueezeNet vs 1D-SqueezeNet training specs

Training Specs	SqueezeNet	1D-SqueezeNet
Iterations/sec	0.8	0.76
Total iterations	510000	510000
No of parameters	96*3*7*7=14112	48*3*7+48*3*7=2016
% reduction in parameters	-	85.71%
Accuracy Top1, Top5	80.59%,56.7%	79.89%,56.45%

Table 5.13: Network wise weight reduction in first layer and increase in total number of weights

CNN	Parameters in First Layer		Parameter increase in	Drop in
	Original Network	1D Network	1D replacement	Accuracy(%)
1D Capsnet V1	31104	3456	3276800	0
1D Capsnet V2	31104	3456	0	0
1D MNIST CNN	288	96	0	0.3
1D CIFAR10 Resnet-18	9408	1344	0	2.6
1D CIFAR10 ZFNET	7200	1440	0	3.97
1D CIFAR100 Resnet-50	9408	1344	0	1.54
1D SqueezeNet	14112	2016	0	0.5
1D Alexnet V2	23328	2592	0	2.3
1D Googlenet*	9408	1344	55296	2.6

As shown in Table 5.13, except in case of Googlenet and Capsnet V1 there is no increase in the number of parameters due to 1D replacement. In case of Capsnet, increase in parameters is due to stride one convolution layer in Capsnet and our approach to trim the input avoided the increase in output size as you can see in Capsnet V2 and in the case of Googlenet, in 1D convolution replacement of Googlenet first convolution layer. i.e 64, 7×7 Convolution with 32, 7×1 and 32, 1×7 caused a large drop in accuracy, but 48, 7×1 and 48, 1×7 convolution replacement had only minimum drop in accuracy. Due to an Increase in the total number of channels in the output of the first convolution layer from 64 to 96. Number of weights in weight matrix of next convolution layer increased from $192 \times 9 \times 64$ to $192 \times 9 \times 96$. This is due to the inability of 32, 7×1 and 1×7 convolutions were not complex enough to capture a similar amount of features represented by original 64, 7×7 convolution layer. As shown in Fig-5.3 and Fig-5.5 Resnet-18 and Resnet-15 networks had minimum drop

in accuracy due to 1D convolution replacement even though they had same 64, 7×7 convolution layer. This is because CIFAR10 [21] and CIFAR100 [32] datasets are not as complex and large as Imagenet dataset [16]. Thus the effectiveness of the number of 1D convolution filters in 1D convolution replacement layer is dependent on the complexity of data that is to be represented of the complexity of feature space to be extracted. Our approach to trim the boundary pixels of input considering them to be less significant is applicable only if the dataset being used to train or test is centered, and most of the boundary pixels are noise And also input trimming approach effectiveness increases as the size of input increases. In case of larger input size, number pixels to be trimmed would proportionately less compared to a small input case.

5.3 Architecture Evaluation

In this section 1D Convolution accelerator design from [1] results are reported. The 1D algorithm solution shows its true benefit when implemented on top of algorithm-aware architecture. In the proposed architecture, spatial parallelism is configurable by the number of MAC units as shown in equation 5.1. For comparison, it is assumed that maximum spatial parallelism resulting in the minimum frequency required, although costing heavy resource utilization.

$$InputDataRate = FPS \times (ImageSize^2) \tag{5.1}$$

$$C_f = \frac{InputDataRate}{ImageSize \times stride} \tag{5.2}$$

$$OperatingFreq = \frac{C_f \times K \times F_{size}^2 \times W_{max}}{K_{macs} \times C_{macs}} \tag{5.3}$$

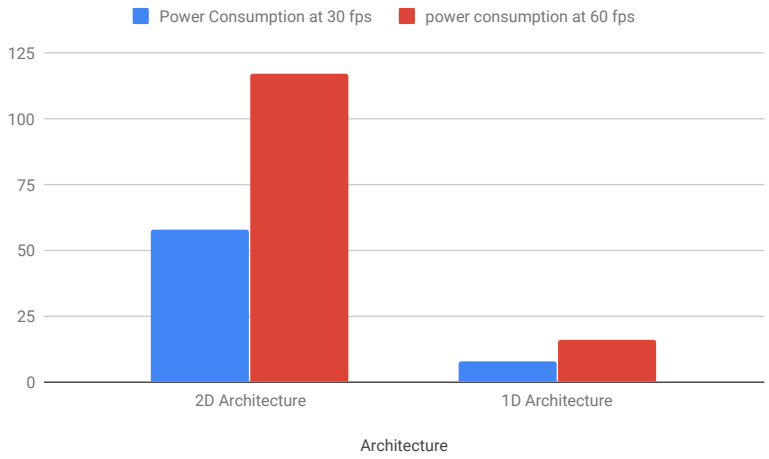


Figure 5.9: Dynamic Power consumption of 2D and 1D architecture at 30 and 60 fps

Fig-5.9 shows an overall power reduction of around 7.3 for both 30 and 60 FPS. This is due to the lower frequency required to meet the constraints of both Frame rates for 1D convolution.

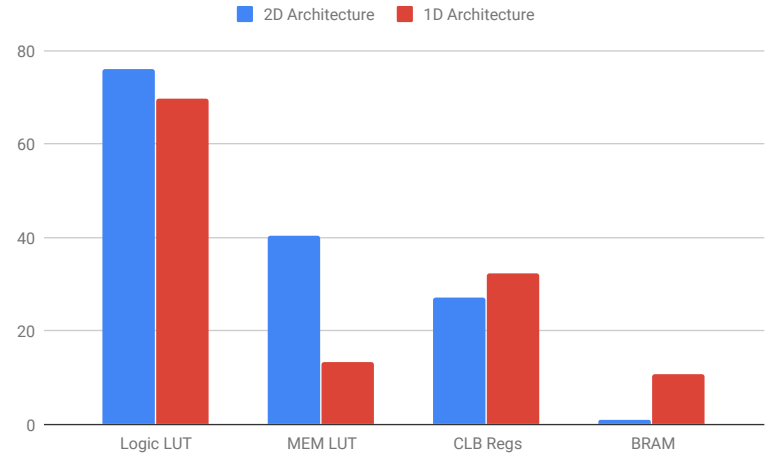


Figure 5.10: Architecute vs Component Resource Utilization

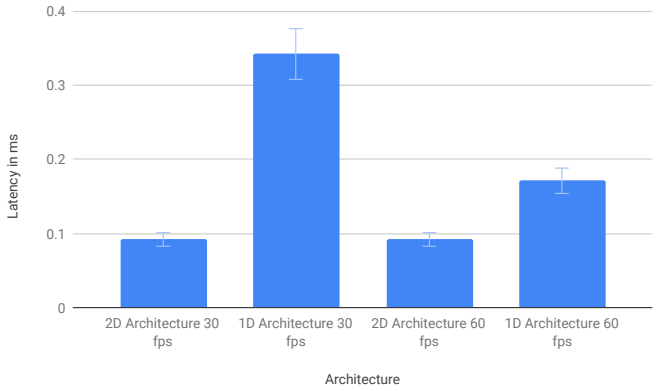


Figure 5.11: Latency comparison between 1D and 2D architectures at 30fps ad 60fps in ms

The resource utilization for both 2D and 1D designs reported in Fig-5.10. It is observed that while both designs utilize a fair amount of resources. The 1D architecture utilizes 6.23% fewer LUTs for logic and 27.04% fewer LUTs for memory with an increase of 5.04% in CLB registers and 9.64% increase in BRAMs. Since the 2D design was more memory intensive, it utilized more LUTs as a memory, but the memory access and control flow of the 1D designed was more optimized, so more resources were translated to BRAMS and CLB registers.

The final metric is the imposed latency on the inference. Since less buffering is required for 1D design it inherently less latency, however, because design runs with a small frequency, it costs more. There is a decrease in latency due to increase of frame rate, due to the significant rise in operation frequency. Furthermore, all designs are magnitudes better than the Jetson TX2 which imposed a latency of 31.4ms, 98.1x greater than our highest latency. The mobile GPU is unable to run at 60 FPS because of this limitation.

CHAPTER 6: CONCLUSION

In conclusion from the results, 1D convolution replacement provides an edge friendly streaming hardware accelerator design with a minimum drop in accuracy. As shown in Fig-5.13 most network architectures showed a minimum reduction in accuracy, and drop in accuracy is dependant on Convolution stride, filter size, and data complexity to be trained. Alexnet results show that Combination of 1D convolution layer followed by rectangular pooling, prevents the increase in the output size, without a drop in accuracy due to 1D convolution in CNN's with the Fully connected layer or replacing FC layer with 1 x 1 convolution and Maxpooling can be used to avoid an increase in size. 1D replacement would be advantageous if the number of filters in a 1D convolution filters is equal to or less than the number of convolution filters in original 2D Convolution layer. In the case of convolution layer with stride 1 produces the same output as after convolution layer preventing the reduction in output size making 1D replacement infeasible with convolution layer with stride one like in the case of Capsnet. For centered datasets like MNIST and CIFAR-10, input trimming approach can be used to avoid the increase in output size and achieve similar accuracy. But in future with application of capsnet on to problem statement involving large and high resolution inputs with feasible stride on convolution,1D convolution replacement would be enormously advantageous as Capsnet model have less hierarchy of features extracted, and number of features extracted in the first layer would be large, as in case simple problem like MNIST and CIFAR 10, 256 filters were required, much complex problems would require more number of filters, multiplying the advantage of 1D replacement.

6.1 Future Work

Even though Capsnet been published in 2017, many of its applications are based on small input size [7],[15],[22]. In [7] Capsnet was used in object tracking but on an image of size 24×24. It is still unclear about the effective use of dynamic routing in case of problems with multiple hierarchies of features with dependencies needing multi-layered routing. Still, many problem statements would be able to utilize dynamic routing in establishing a link between lower level features and higher level features [15]. In case of image detection, for example, face detection CNN, both training and validation inputs would, and for proper prediction with CNNs, CNN need not learn the relation between eyes, nose and ears and face as CNN will never be asked to predict on a disfigured face. But in the case of histology images were the hierarchy of features to be extracted are less and pathological decision or prediction being dependent on feature relation can effectively utilize the dynamic routing. Hence further research on Capsnet to understand multi-staged dynamic routing effect and to optimize the dynamic routing algorithm to be less compute-intensive would be next step.

REFERENCES

[1] J. Sanchez, N. Soltani, R. Chamarthi, A. Sawant, and H. Tabkhi, "A novel 1d-convolution accelerator for low-power real-time cnn processing on the edge," pp. 1–8, 09 2018.

[2] A. Canziani, A. Paszke, and E. Culurciello, "An analysis of deep neural network models for practical applications," *arXiv preprint arXiv:1605.07678*, 2016.

[3] A. Anderson, A. Vasudevan, C. Keane, and D. Gregg, "Low-memory gemm-based convolution algorithms for deep neural networks," *CoRR*, vol. abs/1709.03395, 2017.

[4] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," in *Proceedings of the IEEE*, vol. 86, pp. 2278–2324, 1998.

[5] Nvidia, "Nvidia tesla v100 gpu architecture," 2017.

[6] Nvidia, "Nvidia turing gpu architecture," 2018.

[7] H. Mikami, H. Suganuma, P. U.-Chupala, Y. Tanaka, and Y. Kageyama, "Imagenet/resnet-50 training in 224 seconds," *CoRR*, vol. abs/1811.05233, 2018.

[8] Y. LeCun and C. Cortes, "MNIST handwritten digit database," 2010.

[9] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *CoRR*, vol. abs/1512.03385, 2015.

[10] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems 25* (F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, eds.), pp. 1097–1105, Curran Associates, Inc., 2012.

[11] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *CoRR*, vol. abs/1409.1556, 2014.

[12] J. Sanchez, N. Soltani, P. Kulkarni, R. Vikas Chamarthi, and H. Tabkhi, "A reconfigurable streaming processor for real-time low-power execution of convolutional neural networks at the edge," 06 2018.

[13] O. Ronneberger, P. Fischer, and T. Brox, "U-net: Convolutional networks for biomedical image segmentation," *CoRR*, vol. abs/1505.04597, 2015.

[14] D. Q. Nguyen, T. Vu, T. D. Nguyen, D. Q. Nguyen, and D. Q. Phung, "A capsule network-based embedding model for knowledge graph completion and search personalization," *CoRR*, vol. abs/1808.04122, 2018.

[15] K. Qiao, C. Zhang, L. Wang, J. Chen, L. Zeng, L. Tong, and B. Yan, "Accurate reconstruction of image stimuli from human functional magnetic resonance imaging based on the decoding model with capsule network architecture," in *Front. Neuroinform.*, 2018.

[16] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "ImageNet: A Large-Scale Hierarchical Image Database," in *CVPR09*, 2009.

[17] F. N. Iandola, M. W. Moskewicz, K. Ashraf, S. Han, W. J. Dally, and K. Keutzer, "Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <1mb model size," *CoRR*, vol. abs/1602.07360, 2016.

[18] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," 2015.

[19] M. D. Zeiler and R. Fergus, "Visualizing and understanding convolutional networks," *CoRR*, vol. abs/1311.2901, 2013.

[20] S. Sabour, N. Frosst, and G. E. Hinton, "Dynamic routing between capsules," *CoRR*, vol. abs/1710.09829, 2017.

[21] A. Krizhevsky, "Learning multiple layers of features from tiny images," 2009.

[22] D. Masters and C. Luschi, "Revisiting small batch training for deep neural networks," *CoRR*, vol. abs/1804.07612, 2018.

[23] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015. Software available from tensorflow.org.

[24] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," in *Proceedings of the 22Nd ACM International Conference on Multimedia*, MM '14, (New York, NY, USA), pp. 675–678, ACM, 2014.

[25] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang, "Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems," *CoRR*, vol. abs/1512.01274, 2015.

[26] CS231n, "Convolutional neural networks for visual recognition."

[27] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "Squeezenet: Alexnet-level accuracy with 50x fewer parameters and< 0.5 mb model size," *arXiv preprint arXiv:1602.07360*, 2016.

[28] F. Chollet *et al.*, "Keras," 2015.

[29] BVLC, "Bvlc model zoo."

[30] Nvidia, "Nvidia tesla p100 architecture," 2016.

[31] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, "cudnn: Efficient primitives for deep learning," *arXiv preprint arXiv:1410.0759*, 2014.

[32] A. Krizhevsky, V. Nair, and G. Hinton, "Cifar-100 (canadian institute for advanced research),"