Accelerator Design for Machine Learning

A B. Tech Project Report Submitted
in Partial Fulfillment of the Requirements
for the Degree of

Bachelor of Technology

by

Saurabh Baranwal (180101072)
Amey Varhade 180101087)

under the guidance of

Dr. Hemangee K. Kapoor

to the

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY GUWAHATI
GUWAHATI - 781039, ASSAM

Acknowledgements

We would like to thank our supervisor, Dr. Hemangee K. Kapoor for her support during the course of the BTP. We would also like to thank Ms. Imlijungla Longchar for her invaluable inputs and guidance.

Contents

List of Figures

Chapter 1

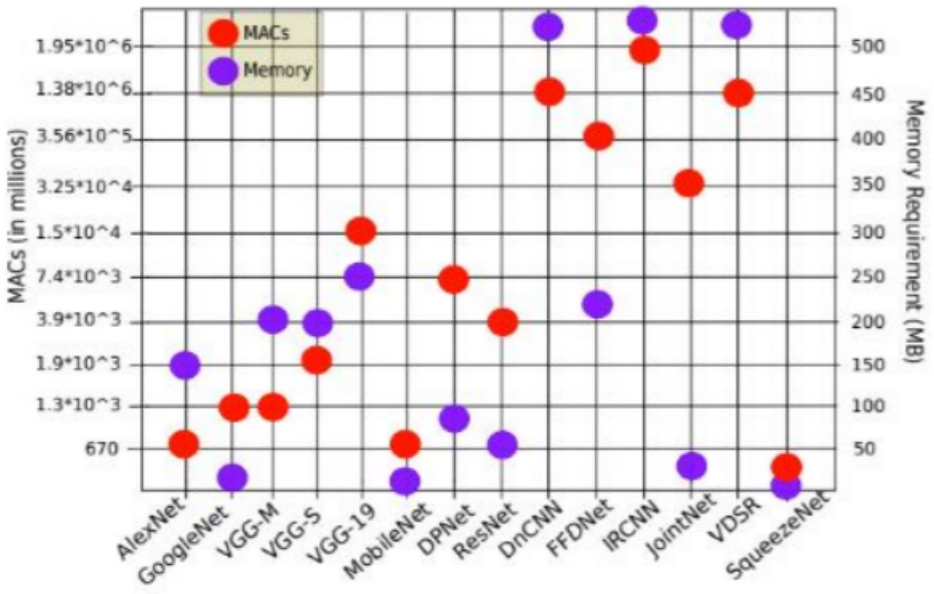
Introduction

Acceleration of hardware involves offloading some computing tasks onto hardware components which are specialized in nature. Compared to a software running on a single CPU, it provides more efficiency.

Machine Learning Accelerators are the specialized hardware components designed to improve the efficacy of machine learning-based applications including but not restricted to computer vision, artificial neural networks, and other data-driven or sensor-driven tasks.

1.1 Motivation and Background

The size of data that needs to be processed is increasing at a very rapid rate in the previous per decade. This has led to an exponential increase in the number of parameters. Hence, the need to efficiently run these networks is urgent. In CNNs, more than half of the computation time is contributed by the convolutional layers. The pooling and Fully Connected layers take relatively less time.



In tasks pertaining to image, speech and vision processing, CNNs have been a very useful technology. Specific challenges for running these algorithms in real time are posed when there is an increment in time and space complexity. Specifically, CNNs have large costs in terms of computation time, number of MAC operations,latency, space complexity, memory, throughput, bandwidth, and power consumption.

1.2 Problem Statement

Accelerator Design for Machine Learning. Reduction of computation time in the inference of CNNs through predictive early activation.

1.3 Organization of The Report

This chapter provided the motivation and background for the topics covered in this report. We also defined the problem statement in this chapter. The organization of subsequent chapters are as follows. In Chapter 2, we review some of the prior works relating to

reduction in memory footprints and computational time of CNNs. In Chapter 3, we propose a greedy algorithm for early prediction of negative computations in CNNs and in chapter 4, we propose a clustering algorithm to reduce the number of operations during a CNN computation. In both chapters 3 and 4, we mention about the motivation, formulation of algorithm, pseudo code, results and limitations of the proposal. Finally in Chapter 5, we conclude our report with some future works.

Chapter 2

Review of Prior Works

To handle the task described, all the pertaining research papers and their methods have been summarized here. These provide an overview of the work done in the field and then we describe our contributions in the subsequent chapters.

The papers have been divided into two categories: Reduction in memory footprints and Reduction in computational time.

2.1 Reduction in Memory Footprint

Large memory in CNNs occur due to intermediate values, high precision of input and filter weights and storage of both effectual and ineffectual data.

2.1.1 Compression of data and weight

Memsqueezer, a memory architecture, was designed by Wang. It has an active weight buffer set, which contains all the weights. Base delta partitioning method enables to store a subset of weights by decomposing them into two values: one mean value (usually base), and the other are a set of differences. We save some bits if numbers are stored as offset values.

Two tables: Delta and Base Table, whose indexing is done via index buffer.

Limitation: It cannot be used for intermediate data in CNNs since the intermediate data changes dynamically. For intermediate data, Wang proposed a data buffer set. For compression, we employ frequent pattern mining method. Identification of patterns which are frequently seen is done and their replacement is done with smaller identifiers, for example Huffman coding. Reduction in area overhead is achieved but some other overheads (computational in nature) may be introduced.

2.1.2 Ineffectual bits reduction in Binary representation

There are two types of ineffectual bits: static and dynamic. Static ineffectual bits are those bits which might not be necessary for representation. Dynamic ineffectual bits are the zero bits that are necessary for representation but are useless with respect to computation operations (like multiplication). Eg: 0010010 has 2 dynamic ineffectual bits and 3 static ineffectual bits.

2.1.2.1 Removal of static ineffectual bits

STR Proposal. Precision of each CNN layer is defined as maximum static effectual bits required to store all activations for that layer. Since each layer would've required a different multiplier unit, they introduced a serial multiplier unit. It increases multiplication latency by serial data bit-width (b) , thus b such parallel units are used which improves performance by f/b (where f is fixed bit-width). Also, desirability of precision for each layer is different, buffer size at output can be customized to save further space.

The precision of a group of activations at runtime is determined through dyamic STR. Thus, only those activations are considered that are currently being executed. Dynamic approaches are usually more effective but they have more overheads.

2.1.2.2 Removal of dynamic ineffectual bits from activations

Pragmatic exploits dynamic ineffectuality. Mahmoud further proposed another extension over pragmatic by another convolution operation. For windows other than the first one, differences of pixel values with respect to first window are stored. This reduces memory. In precision, computation involving dynamic ineffectual bits are not calculated. This improves the latency further.

2.1.3 Removal of ineffectual bits from weights

We also use bit multipliers for this purpose. LOOM exploits serial processing for both activation and weights, unlike previously which only exploited it for activation and had fixed the weight bit width. Thus, latency may be more but memory used is less (since operations are performed bit by bit for both activation and weights).

2.1.4 Optimizing performance of effectual bits

Bit serial multipliers have a limitation; hence a different architecture is proposed. 2*2 multiplier units, called Bitbrick, are used. It can be fused with other bitbricks to create larger multipliers.

How to fuse databricks?

Spatial: An n bit multiplication can be broken down into several 2*2 bitbricks

Temporal: Multiple multiplications can be performed on same cluster of bitbricks at different times.

LUT

Assume that you have 1 bit activations: X,Y,Z. Now, we multiply them with weights wx,wy,wz (MSB: LSB). There will be 8 combinations of wx,wy,wz and hence they are stored in a Lookup Table for reuse.

Details such as the number of memory buffers and the nature of the connections are very important to performance. Although there is an increment in storage requirements, it is accompnied by reduction in the amount of computation. Clearly we need to balance the tradeoff between space and time here.

2.2 Reduction in Computational Time

2.2.1 Exploiting Inherent Parallelism

There are nested loops, we can pick up any axes and parallelize them specifically out of the 6 available axes.

2.2.2 Pattern based Computation reduction

Number of unique weights is constrained by the maximum number of bits. Here we exploit the associativity in multiplication. There is a time vs space tradeoff.

2.2.3 Cavoluche [Wang et. al.]

This is a software and hardware based solution which uses Principal Component Analysis (PCA) to reduce the size of the weights and stores the intermediate results. Implementation involves NVIDIA TX1 vs RTL synthesized Cavoluche (Synopsys Design Compiler, 65nm TSMC).

2.2.4 Reducing inefficient computations

Pruning values below a certain threshold (or zero) helps in efficient computation. It saves both time and energy. Redundancy filters are used to remove the ineffectual computations.

2.2.5 Prediction Driven Computation Reduction

Computations filtered by ReLu and the Pooling layer are ineffectual. Different techniques for their prediction have been proposed.

2.1.5.1 Prediction based execution [Song et. al.]

High order bits are passed through the comparator and sign unit to predict computations rendered ineffective by pooling and Relu layers. Predicted computations are marked as zero in both tables. But this doesn't reduce computation time but only reduces power (as PEs are parallel and this leads to only idleness of PEs as one PE sits idle till its neighbouring PEs complete their computation).

How to reduce computation time?

Allow data fetching in only one direction (either activation or weights). Data from other directions can be fetched from memory. This reduces computation time but increases pressure on the memory system.

2.1.5.2 SnaPEA

We rearrange weights in such a way as to know if the computation overall results are negative at the earliest. This paper proposes a method that leads to reduction of those computations by taking into consideration both algorithmic structure and intformation about runtime of CNNs. Predictive Early Activation Technique has two modes, namely Exact mode and Predictive mode.

These cut the computation of convolutional operations short if it predicts the final output of a computation to be negative during runtime. In this way, we prune the ineffectual computations.

1. Perform computation of add and multiply with positive weights
2. Sort the negative weights in decreasing order of their magnitudes

3. In each cycle, perform computation using one negative weight in order. If at any stage, overall computation becomes negative then terminate.

The above mode is known as the **exact mode.**

Predictive mode

Threshold value predetermined along with no. of MAC operations for that threshold. Weights are divided into 3 classes: predictive, positive and negative. Computations are done on predictive weights and when compared with a threshold, we give a prediction if computation will be ineffective or not.

Some false positives may occur. The advantage is that computations are reduced to size K*K - N where N is the count of predictive weights. Since the input values could be random, we're ignoring their contribution when we select larger magnitude weights. SnaPEA exact mode may not be effective for models where there are many negative weights with similar magnitude.

$$\mathrm{Op}(o_{l,k}^{d},\mathrm{Th}_{l}^{k},\mathrm{N}_{l}^{k})=\begin{cases}\mathrm{N}_{l}^{k}, & \text{if } \mathrm{PartialSum}_{\mathrm{N}_{l}^{k}}\leq \mathrm{Th}_{l}^{k},\\ \mathrm{Idx}_{w^{-}}, & \text{if } \mathrm{PartialSum}_{\mathrm{N}_{l}^{k}}>\mathrm{Th}_{l}^{k} \textit{ and } \mathrm{PartialSum}_{w^{-}}\leq 0,\\ \mathrm{C}_{in,l}\times \mathrm{D}_{l}^{k}\times \mathrm{D}_{l}^{k}, & \text{otherwise}\end{cases}$$

$$\min_{\mathrm{Th,N}} \sum_{d\in\mathscr{D}}\sum_{l\in L}\sum_{k\in K_l}\sum_{o\in O_{l,k}^{d}} \mathrm{Op}(o,\mathrm{Th}_{l}^{k},\mathrm{N}_{l}^{k})$$

$$\text{subject to} \quad \mathrm{Accuracy}_{CNN}-\mathrm{Accuracy}_{SnaPEA}\leq\varepsilon$$

Fig. 2.1 Optimisation Problem for SnaPEA

Sort the weights in increasingly, partition them into a number of smaller groups, and select the weight with the largest magnitude from each group (called predictive weights)

Then keep the remaining positive and negative weights in order. Computations are first

done on predictive weights and then compared with a threshold value. If less than the threshold, predict negative output and terminate.

If greater than the threshold, do further computations overall positive values and then over negative values. If at any negative weight index, partial computation comes negative, terminate. Else, terminate after computing for every weight value.

Note: SnaPEA model may not be effective for models where there are many negative weights with similar magnitude.

Implementation involves GPU and Custom hardware synthesized on TSMC 45nm standard cell library using Synopsys Design Compiler (SP5).

2.3 Conclusion

The research papers explored, provide insights into improvements, using accelerator design, in reducing computational time and memory footprints for CNNs.

At the expense of accuracy, we are achieving higher performance and reduction in memory footprint. We need to balance this tradeoff well. For CNNs we can make these adjustments according to layers or type of layers.

Chapter 3

Proposed Algorithm I

In this chapter, we propose a greedy approximate algorithm to predict whether a given CNN computation overall results negative at the earliest, after a given set of operations.

3.1 Motivation

In the exact mode of SnaPEA, it was seen that the greedy strategy of sorting negative values in decreasing order of their absolute values allowed to cut short operations with respect to some weights.

As an extension to the above greedy strategy, we devise a greedy strategy that involves sorting of both positive and negative weights in decreasing order of their absolute values. The motivation behind this idea is that the contribution corresponding to most negative value shall be neutralised out by the contribution corresponding to most positive value.

3.2 Formulation of Algorithm

1. Sort both positive and negative weights separately, in decreasing ordered of their absolute values. We maintain a partial sum initialised to zero, for CNN computation as we progress through.

2. Iterate through the sorted positive and negative weights list until one of them finishes. At each iteration i, add the contribution to partial sum corresponding to both positive and negative weights stored at index 'i' in respective sorted lists.

3. If at any iteration the partial sum becomes negative, we predict that the overall computation shall be negative.

As discussed before, the motivation behind this idea was that the most positive values shall neutralise the contribution of most negative values in the overall computation. Hence the sorting strategy was based on this criterion. The assumption we make while terminating as soon as partial sum becomes negative, is that since the most positive values were not able to neutralise the effect of most negative values hence the overall computation is predictably negative.

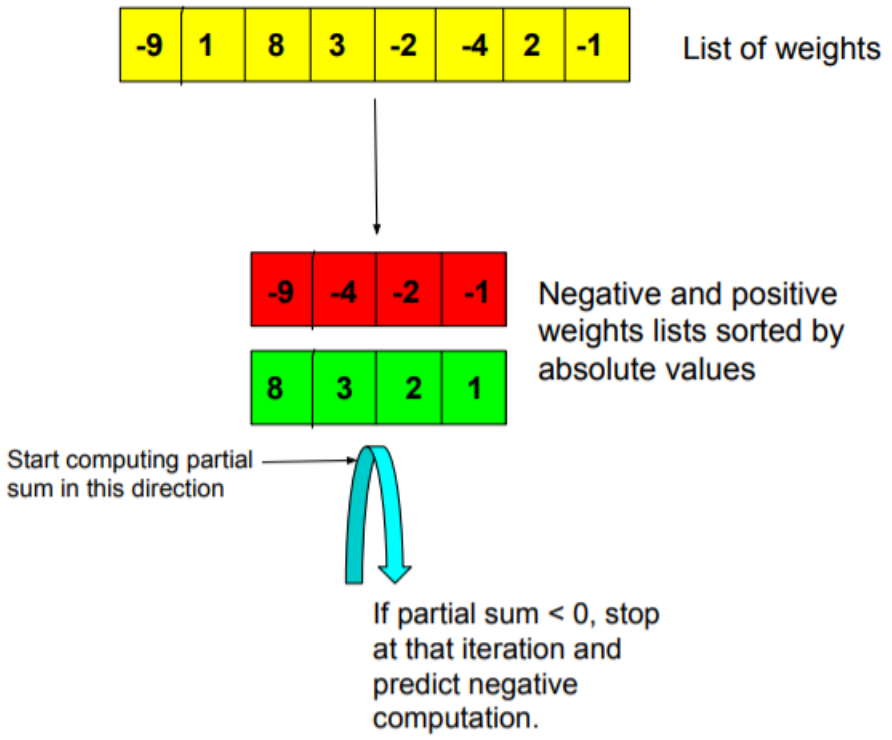


Figure: Proposed Algorithm 1

3.3 Pseudo Code of the proposed algorithm

Algorithm 1 Greedy Algorithm

```
W {} ← List of weights
pos = {} ← List of positive weights
neg = {} ← List of negative weights
for weight w in list W do
    if w < 0 then neg.add(w)
    if w > 0 then pos.add(w)
Sort pos in descending order
Sort neg in ascending order
x ← minimum (size of pos, size of neg)
idx ← 0
partialSum ← 0
while idx < x do                    ▷ Iterate through lists using while loop and index idx
    positive_contribution = pos[i] * (corresponding input value)
    negative_contribution = neg[i] * (corresponding input value)
    partialSum+ = (negative_contribution + positive_contribution)
    if partialSum < 0 then
        Report overall computation as negative
        break loop
    else continue
    idx ← idx+1
end while
```

3.4 Results

We wrote the Python codes for original CNN computation, SnaPEA exact mode and for our proposed algorithm. In this section, we try to compare the intermediate results from the prototype of our ideas on a particular layer.

3.4.1 Comparison of image

For our original CNN computation, we generate an image using standard Gaussian distribution. We then apply modifications of SnaPEA exact mode and our proposed algorithm to see the difference between the images reported. The results were as follows:

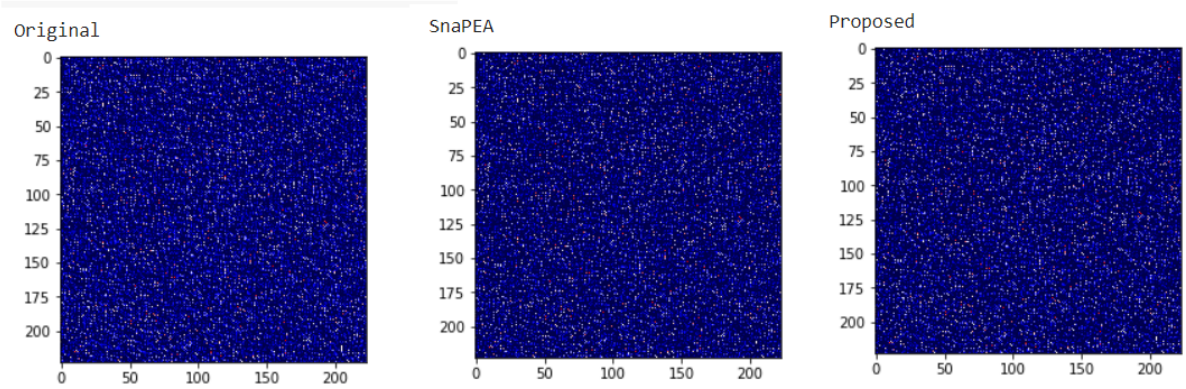


Fig. 3.1 Comparison of images generated

3.4.2 Number of multiplication operations

In our Python codes for respective cases, we maintain count for total multiplication operations performed in each case. A short snippet and output of the code is as follows:

```
[22] print("Snapea: ", snapea_count)
     print("Proposed Algorithm: ", proposed_algo_count)
     print("Original count: ", original_count)

     Snapea:  283300
     Proposed Algorithm:  108240
     Original count:  447561
```

Fig. 3.2 Comparison of operations

We see that in original CNN computation, total operations were 447561. In SnaPEA exact mode, there are 283300 operations and in our proposed algorithm, total operations are 108240. We conclude that the SnaPEA exact mode and our proposed algorithm significantly reduce the number of operations.

3.4.3 Precision, Recall and F1 Score

In both SnaPEA and our proposed algorithm, there are wrongly predicted false positives (FP) , false negatives (FN) and the correctly predicted true positives (TP) and true neg-

atives (TN). A good metric for comparison would be to calculate these values to find respective precision, recall and F1 score for each case. We do the same in our Python code.

Mathematically precision, recall and F1 Score are defined as follows:

- Precision = (TP)/(FP + tP)
- Recall = (TP)/(FN + TP)
- F1 Score= (2 * (Recall * Precision)) / (Precision + Recall)

Firstly, we calculate values of FP, FN, TP and TN for both SnaPEA and proposed algorithm using our Python code. Then we report the precision, recall and F1 score for both cases. A snippet from our code, showing it is as follows:

```
[23] snapea_precision = S_TP/(S_TP + S_FP)
     snapea_recall = S_TP/(S_TP + S_FN)

     prop_precision = Pr_TP/(Pr_TP + Pr_FP)
     prop_recall = Pr_TP/(Pr_TP + Pr_FN)

     print("Snapea Precision : ", snapea_precision)
     print("Snapea Recall : ", snapea_recall)

     print("Proposed Algorithm Precision: ", prop_precision)
     print("Proposed Algorithm Recall: ", prop_recall)

     S_F1 = 2*(snapea_recall*snapea_precision)/(snapea_precision+snapea_recall)
     Pr_F1 = 2*(prop_recall*prop_precision)/(prop_precision+prop_recall)

     print("Snapea F1 Score: ", S_F1)
     print("Proposed Algorithm F1 Score: ", Pr_F1)

     Snapea Precision :  1.0
     Snapea Recall :  0.8386917147797543
     Proposed Algorithm Precision:  0.9962959080266974
     Proposed Algorithm Recall:  0.8441450777202073
     Snapea F1 Score:  0.9122700755522968
     Proposed Algorithm F1 Score:  0.913931273240159
```

Fig. 3.3 Precision, Recall and F1 Score

We see that the F1 Score of SnaPEA comes out to be 0.9122700 and the F1 score of our proposed algorithm comes out to be 0.9139312. Both are almost comparable.

3.5 Limitations

Since the proposed algorithm is greedy and approximate in nature, it may not necessarily predict negative computations correctly every time. Hence, some false positives are bound to occur.

These false positives can occur either due to variations in input values (as our algorithm involved no computation on activation values) or due to variations because of weights having lower absolute values contributing more.

Chapter 4

Proposed Algorithm II

In this chapter, we propose a clustering algorithm to reduce the number of operations in a CNN computation. The algorithm being proposed shall employ standard 1-D K-means clustering and we shall demonstrate how it reduces the number of operations for CNN computations. This shall result in reduction of computation time as well.

4.1 Motivation

The motivation to devise this algorithm came through the predictive mode of SnaPEA, where we were dividing weights into k groups and choosing the weight having maximum absolute value as the predictive weight.

The algorithm we propose involves the division of weight list into some k clusters and choosing representative element from each cluster via the K-means approach. All the weights in respective operations shall be replaced with their representative cluster means and we shall see how it reduces the number of operations.

4.2 Formulation of Algorithm

1. Make a good choice of K for the given weight list. K denotes the number of clusters in which our weight list shall be divided.

2. Using K-means approach, we divide our weight list into K clusters and find K cluster means. Each cluster mean shall act as the representative weight for the entire cluster.

3. In every operation while doing CNN computation, replace the weights with their respective cluster means. We shall see mathematically that in this way, we save significant amount of time to perform those operations.

Let us demonstrate it mathematically through an example.

Let us consider 9 weights in our weight list {w1,w2,w3,w4,w5,w6,w7,w8,w9}, the corresponding input values to be {a1,a2,a3,....,a8,a9} and number of clusters to divide them to be 3.

Using K means approach, we divide them in 3 clusters and find the respective cluster means.

Cluster 1: {*w1, w2, w7*} , **Mean 1:** m1

Cluster 2: {*w3, w6, w9*} , **Mean 2:** m2

Cluster 3: {*w4, w5, w8*} , **Mean 3:** m3

Case 1: Before forming clusters, output value = $\sum_{n=1}^{9} w_i a_i$

Case 2: After forming clusters and replacement by the means, output value = *m1*(a1+a2+a7) + m2*(a3+a6+a9) + m3*(a4+a5+a8)*

We see that total addition operations remain 8 in both cases, but total multiplication operations reduce from 9 in first case to 3 in second case.

4.3 Proposed algorithm

Algorithm 2 Adaptive 1-D K-means clustering algorithm

1: Assign the total number of clusters, k.
2: Initialise k means randomly.
3: **repeat**
4: **expectation:** Assign each point to closest mean.
5: **maximization:** Find the new centroid mean corresponding to each cluster.
6: **until** Mean positions become constant (no change)

Algorithm 3 Proposed algorithm

1: Apply 1-D K means clustering on weight list
2: Maintain a list of cluster means and weights in respective clusters
3: *sum*= 0
4: i = 0 (variable for index of cluster)
5: **repeat**
6: *sum* += (ith cluster mean)*(sum of input values of ith cluster)
7: i= i+1
8: **until** i becomes K
9: Report *sum* as final CNN computation value

4.4 Results

We wrote the Python codes for original CNN computation, SnaPEA exact mode and for our proposed clustering algorithm. In this section, we try to compare the results that we got.

4.4.1 Comparison of image

For our original CNN computation, we generate an image using standard Gaussian distribution. We then apply modifications of SnaPEA exact mode and our proposed clustering algorithm to see the difference between the images reported. The results were as follows:

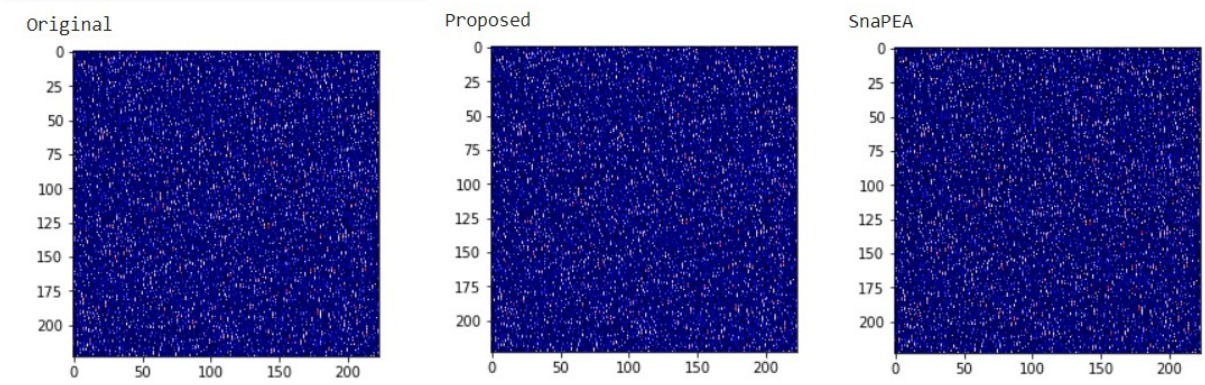


Fig. 4.1 Comparison of images generated

4.4.2 Number of multiplication operations

In our Python codes for respective cases, we maintain count for total multiplication operations performed in each case. A short snippet and output of the code is as follows:

```
[ ]  print("Snapea: ", snapea_count)
     print("Proposed Algorithm: ", proposed_algo_count)
     print("Original count: ", original_count)

     Snapea:  285619
     Proposed Algorithm:  149187
     Original count:  447561
```

Fig. 4.2 Caption

We see that in original CNN computation, total operations were 447561. In SnaPEA exact mode, there are 285619 operations and in our proposed algorithm, total operations are 149187. We conclude that the SnaPEA exact mode and our proposed clustering algorithm significantly reduce the number of operations. This was because elements of each cluster were replaced by their cluster mean, which led to reduction of several multiplication operations.

4.4.3 Precision, Recall and F1 Score

In both SnaPEA and our proposed clustering algorithm, there are false positives (FP) , false negatives (FN) , true positives (TP) and true negatives (TN). A good metric for comparison would be to calculate these values to find respective precision, recall and F1 score for each case. We do the same in our Python code.

We've already described mathematical formula for each of them in previous section. Hence, here also we calculate precision, recall and F1 Score for our clustering algorithm. They were reported by the output of code as follows:

```
snapea_precision = S_TP/(S_TP + S_FP)
snapea_recall = S_TP/(S_TP + S_FN)

prop_precision = Pr_TP/(Pr_TP + Pr_FP)
prop_recall = Pr_TP/(Pr_TP + Pr_FN)

print("Snapea Precision : ", snapea_precision)
print("Snapea Recall : ", snapea_recall)

print("Proposed Algorithm Precision: ", prop_precision)
print("Proposed Algorithm Recall: ", prop_recall)

S_F1 = 2*(snapea_recall*snapea_precision)/(snapea_precision+snapea_recall)
Pr_F1 = 2*(prop_recall*prop_precision)/(prop_precision+prop_recall)

print("Snapea F1 Score: ", S_F1)
print("Proposed Algorithm F1 Score: ", Pr_F1)
```

```
Snapea Precision :  1.0
Snapea Recall :  0.854701698933228
Proposed Algorithm Precision:  0.9209522708887091
Proposed Algorithm Recall:  0.9211651832158132
Snapea F1 Score:  0.9216594770197581
Proposed Algorithm F1 Score:  0.9210587147480351
```

Fig. 4.3 Precision, Recall and F1 Score

We see that the F1 Score of SnaPEA comes out to be 0.9216594 and the F1 score of our proposed clustering algorithm comes out to be 0.921050. Both are almost comparable.

4.5 Limitations

Although there is a reduction in the number of operations, it gives us an approximate answer. Thus, there can be loss in accuracy since we are ignoring individual contributions and only taking contributions of respective representative cluster means.

Thus, the approximate answer that we get may not be close to the actual answer in some cases.

Chapter 5

Conclusion and Future Work

5.1 Conclusion and results

In both our proposed algorithms, we were able to achieve significant reduction in multiplication operations. We compared our performance with against the baseline as SnaPEA exact mode.

The detailed results have already been documented as part of two previous sections. We compile them here in a tabular format :

Model	Operations	F1 Score
SnaPEA Exact Mode 1	283300	0.9122
Proposed Algorithm 1	108240	0.9139
SnaPEA Exact Mode 2	285619	0.9216
Proposed Algorithm 2	149187	0.9215

5.2 Future Work

All the algorithms that we reviewed and proposed as part of this report, involved modifications in weights but no modifications in input values. Since none of the algorithms reviewed/proposed accounts for variation in input values, it would be a nice future work

to do some improvisation in this regard.

In our first proposed algorithm, the reported correctness upto which our algorithm predicts negative computations correctly is subjected to further improvement. With further improvements in this greedy algorithm as a future work, we shall look into how we can improve the accuracy of correct prediction.

In our second proposed algorithm, we used K-means approach to cluster weights. As a future work, we may look into some other mathematical techniques to cluster one dimensional data and compare the results of those approaches with the results of K-means approach.

References

[AYS+18] Vahideh Akhlaghi, Amir Yazdanbakhsh, Kambiz Samadi, Rajesh K. Gupta, and Hadi Esmaeilzadeh. Snapea: Predictive early activation for reducing computation in deep convolutional neural networks. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 662–673, 2018.

[DJS+18] Alberto Delmas, Patrick Judd, Dylan Malone Stuart, Zissis Poulos, Mostafa Mahmoud, Sayeh Sharify, Milos Nikolic, and Andreas Moshovos. Bit-tactical: Exploiting ineffectual computations in convolutional neural networks: Which, why, and how. *CoRR*, abs/1803.03688, 2018.

[DJSM17] Alberto Delmas, Patrick Judd, Sayeh Sharify, and Andreas Moshovos. Dynamic stripes: Exploiting the dynamic precision requirements of activation values in neural networks. *CoRR*, abs/1706.00504, 2017.

[JAH+16] Patrick Judd, Jorge Albericio, Tayler Hetherington, Tor M. Aamodt, and Andreas Moshovos. Stripes: Bit-serial deep neural network computing. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12, 2016.

[LKK+19] Jinmook Lee, Changhyeon Kim, Sanghoon Kang, Dongjoo Shin, Sangyeob Kim, and Hoi-Jun Yoo. Unpu: An energy-efficient deep neural network accelerator

with fully variable weight bit precision. *IEEE Journal of Solid-State Circuits*, 54(1):173–185, 2019.

[PRM+17] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel Emer, Stephen W. Keckler, and William J. Dally. Scnn: An accelerator for compressed-sparse convolutional neural networks, 2017.

[SLS+18] Sayeh Sharify, Alberto Delmas Lascorz, Kevin Siu, Patrick Judd, and Andreas Moshovos. Loom: Exploiting weight and activation precisions to accelerate convolutional neural networks. In *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, pages 1–6, 2018.

[SZH+18] Mingcong Song, Jiechen Zhao, Yang Hu, Jiaqi Zhang, and Tao Li. Prediction based execution on deep neural networks. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 752–763, 2018.

[WLL16] Ying Wang, Huawei Li, and Xiaowei Li. Re-architecting the on-chip memory sub-system of machine-learning accelerator for embedded devices. In *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–6, 2016.

[WLLL19] Ying Wang, Shengwen Liang, Huawei Li, and Xiaowei Li. A none-sparse inference accelerator that distills and reuses the computation redundancy in cnns. In *Proceedings of the 56th Annual Design Automation Conference 2019*, DAC '19, New York, NY, USA, 2019. Association for Computing Machinery.

[]