THESIS

PRUNING AND ACCELERATION OF DEEP NEURAL NETWORKS

Submitted by

Janarthanan Thivagara Sarma

Department of Electrical and Computer Engineering

In partial fulfillment of the requirements

For the Degree of Master of Science

Colorado State University

Fort Collins, Colorado

Spring 2020

Master's Committee:

Advisor: Louis-Noël Pouchet

Sanjay Rajopadhye
Sudeep Pasricha
Chuck Anderson

ABSTRACT

PRUNING AND ACCELERATION OF DEEP NEURAL NETWORKS

Deep neural networks are computational and memory intensive applications. Many network pruning and compression solutions has been introduced to deploy inference of large trained models in limited memory and time critical systems. We proposed a new pruning methodology that assigns significance rank to the operations in the inference program and for a given capacity and operation budget, generate only the important operations to do the inference. Our approach has shown that, in many classical feed forward classification networks we can maintain almost the same accuracy as the original inference by executing less than half of the operations in the original program. We also proposed a methodology to improve the effective implementation of the output sparse computation, controllable by a threshold variable.

ACKNOWLEDGEMENTS

I would like to first thank my advisor Professor Louis-Noël Pouchet for his guidance, support and encouragement that made completion of this thesis possible and I am also grateful to my advisor for the many invaluable advice for my professional and personal life. I also like to thank Professor Sanjay Rajopadhye for his guidance throughout my Master's degree.

I would also like to thank my committee members Professor Sanjay Rajopadhye, Professor Sudeep Pasricha and Professor Chuck Anderson for serving on the defense committee and reviewing this thesis document.

I am using this opportunity to express my gratitude to everyone who supported me throughout my Master's program.

DEDICATION

I would like to dedicate this thesis to my parents.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Background

1.1 Artificial Neural Networks

Artificial neural networks are computational models inspired from biological neural networks[1]. A biological neuron or brain cell receives electric signals transmitted from other neurons, modify them and transmit them to their connections. The input connections to the neuron's cell body are represented by the dendrites and the output from the cell body are represented by the axons. Similarly, an artificial neuron receives its input from other artificial neurons and modify the value and send the output to the next neurons. Fig.1.1 shows a biological neuron and an artificial neuron.

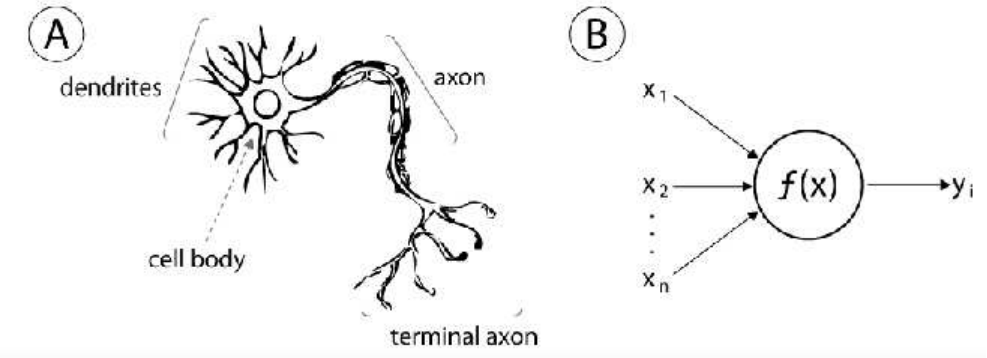


Figure 1.1: (A) Brain cell and a (B) Artificial neuron (image from [1])

Artificial neurons are the basic building blocks of many artificial neural networks. Similar to brain cells, Network of Artificial neurons can be trained using supervised learning methods. With enough number of training information consist of inputs and outputs, the neural network can be learned to approximately map the input data to the expected output.

Artificial neural network is a collection of artificial neurons connected to perform a specific task. Collection of neurons forms a layer in the neural network. A typical neural network contains an input layer, one or more hidden layers and an output layer. Fig.1.2 represents a neural network

with 2 hidden layers. Input layer represents the initial input data for the network, output layer represents the results of the neural network for the given inputs and the hidden layers perform the computations necessary to produce the output layer.

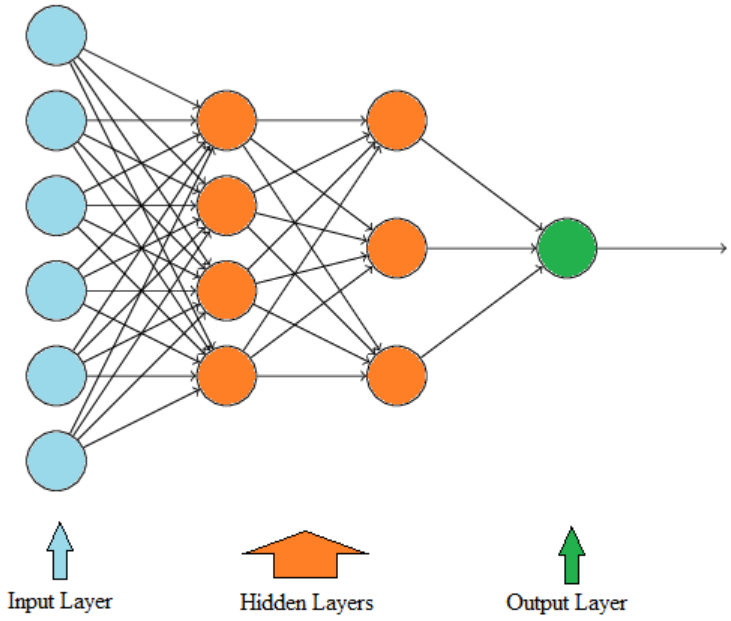


Figure 1.2: A Neural Network with 2 hidden layers

The choice of number of hidden layers and hidden units depends on the problem the Neural Network tries to address. Having larger network enables the neural network's ability to learn to solve complex task, and having too large network will hinder the neural network's generalizability when a new unprecedented data is given to solve. Neural networks can be classified into different categories based on how the neurons are connected. Depending on the problem, some artificial neural networks better suited to solve the problem than others. Section 1.1.1 discusses commonly used categories of artificial neural networks.

1.1.1 Types of artificial neural networks

Classical feed forward neural networks

Feedforward neural networks or multi-layer perceptrons are class of artificial neural networks where the information flow through the network without any feedback connections. Classical feedforward networks are composed of multiple fully connected layers. In a fully connected layer, each input neuron is connected to each of the output neurons.

Cascade Neural Networks is a class of feed forward neural network, where input layer and each previous layer's outputs are connected to the output of the current layer. In this work only classical feed forward neural networks and cascade neural networks are considered for pruning.

Convolution neural networks

Convolutional neural networks are feed forward neural networks where one or more layers in the network are composed of convolutional layers. Convolutional layer can be trained to extract space invariant information from the inputs, and commonly used to analyze visual data. Unlike fully connected layers, each neuron in the convolutional layer is connected to a small region of the input. This connection enables the network to capture spatial local patterns in the input.

Recurrent neural networks & LSTM

Recurrent neural networks (RNN) able to learn to solve problems where the input size is arbitrary. The RNN enables to pass the information from previous steps to current step. This makes the network to remember things learned from previous inputs and retain information through time, which enables to analyze time-series information. LSTM (Long Short Term Memory) is an advancement of RNN, which is used to solve problems requiring to learn long term temporal information. It incorporates memory cells to hold information and gates to control when the information is stored, used and forgotten.

1.1.2 Motivation for approximate computing of inference

The process of training a neural network for a task is, start by choosing the right type of network to trained. Some types of networks are well suited for certain tasks, for example classifying an input image based on some features of the input images could be a good fit for convolutional neural network. Recurrent neural network or long short-term memory is used for task that requires to remember the past decisions when making new decisions or tasks receives a temporal sequence data as input, where information about previous input is necessary to decode current input, like speech audio input in speech recognition. Once the network is chosen, next process is defining the number of parameters to train the network. The larger the parameter space the more flexible the network is to learn complex functions, but it may lead to network over-fitting to the training data and perform poorly when a new data is presented. So selecting the parameter size is important to get better accuracy on actual input data after the network is trained. During the training process, the network classifies the training input data, calculates prediction error and adjust the network's parameters thorough back-propagation to minimize that error, so that the network will be able to minimize the prediction error next time. During the training phase, network learns which parameters or connection between neurons are important, and which are not important.

Approximate computing in classification problems

In classification problem the output variable is a class predicted from a discrete number of classes. Examples of classification problems solved using machine learning are classify an image of handwritten digit between 0-9, classifying a message as spam or not spam etc. All these classification problems select the best category from a list of possible categories. During the learning process, the neural network is trained to classify the input in to correct category, by maximizing the probability of the target category. The network connections or the weight parameters are updated through backpropagation during the training process to learn the input to target relationship. A SoftMax layer is commonly used to convert a vector of output results to a vector of probabilities. In classification neural networks, the index of the highest probability output (argmax) is more relevant than the absolute final output itself. Thus, classification problems are tolerable to approximate

answers, as long as approximation does not change the highest probability class. This makes the classification problems a good candidate for approximate computing.

1.2 Related work

Inference of many trained networks can be accelerated via reducing the size of the network by removing the redundant computations in the network. In many cases, connections between the neurons can be removed given the connection does not provide significant enough contribution to the prediction. In other cases, the network structure can be represented by a more compact format to represent approximated network structure. Researchers also exploiting redundancy in the representation of bits in network weights, in order to reduce the network size and accelerate.

Pruning

Weight pruning has been used to reduce the neural network size for inference and reduce over-fitting to make the network better at generalization. LeCun et al. [3] pruned weights that has less impact on training error estimated using the second derivative of the training objective function. Once the low impacting weights are removed retrain the network and the procedure is repeated. Han et al. [4] initially trained the network to learn the important connections first, then pruned the connections below a threshold and finally retrain the network to gain back the accuracy. Through pruning and retraining a network compression of 9x - 13x was achieved on VGGNET and AlexNet without loss in accuracy. Polyak et al. [5] prune the insignificant channels in the convolutional layers which does not contributes to relevant information extracted.

Structural sparsity

Regular pruning of connections in neural networks will result in sparse structure that will perform poorly due to the unstructured connections. Alternative training methods have been used to overcome the implementation issues of sparse structures. Wen et al. [6] proposed structured sparsity learning to regularize the neural network parameters such as filters and channels to improve the performance of the deployed network. Liu et al. [7] regularized the channels through imposing

a scaling factor on the channel output and pruned the channels with the smaller trained scaling factors. Li et al. [8] pruned the convolutional kernels with smaller sum of absolute kernel weights to drop less important filters and retrained the network.

Layer decomposition

Layer decomposition exploits the redundancy in the network by low rank approximation. Sainath et al.[9] proposed low rank matrix factorization to reduce the number of parameters in the large final layer of speech recoginition networks to improve the training speed. Denton et al. [10] used low rank approximation to exploit the redundancy in the convolution kernels and reduced the number of parameters.

Bit representation

Studies have shown efficient training and implementation of Neural Network on lower precision environments than the traditional single precision floating point. Gupta et al. [11] proposed 16 bit fixed point with stochastic rounding for training on CIFAR 10 dataset. Wang et al. [12] trained 8 bit floating point models of ResNet, AlexNet while maintaining the accuracy. Han et al. [13] proposed a compression method, which prunes the less important connections and then train the remaining connections. This was followed by quantization of connections to compress even further and retrain the quantized network again to gain the accuracy back.

1.2.1 Our Approach

In this work we proposed a methodology to accelerate a fully connected deep neural network inference by executing only the significant operations. For a given pre-trained network, we rank all the operations in the inference program by their importance or significance to the output accuracy, and skip the insignificant operations by executing the high ranked operations only. Since this method executes only the operations that are significance to the output prediction, executing subset of operations leads to smaller inference network with little or no accuracy loss.

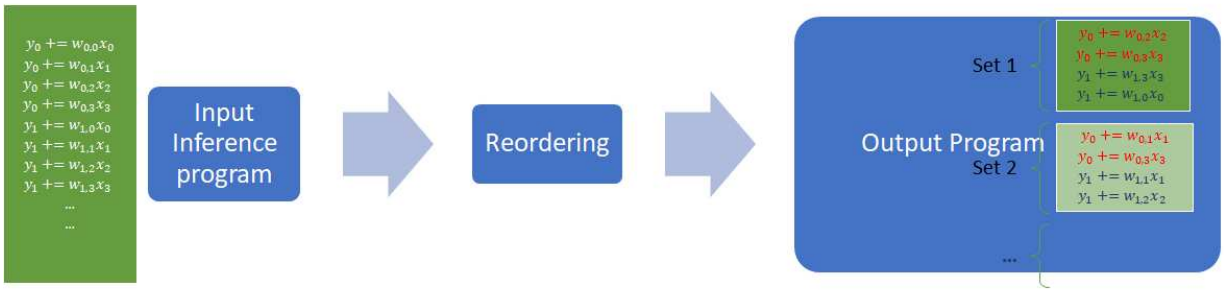


Figure 1.3: Framework overview

Fig.1.3 shows the overview of the proposed methodology where the input inference program is reordered into set of operations where Set 1 have operations higher in rank than Set 2 and Set 2 operations are higher in rank than the operations from Set 3 etc. Depending on the deployment the user has the flexibility to specify the number of operations to be executed in the output program. Chapter 2 discuss about the ranking of operation by significant contribution and producing sets of operations based on the rank. Chapter 3 discuss about generating a better vectorizable program from the program generated from Chapter 2, which leads to better implementation.

Chapter 2

Network Pruning

2.1 Overview of Pruning

2.1.1 Purpose

When training deep neural networks for a classification task, the accuracy of the neural network prediction varies depending on the neural network structure chosen. Choosing a very deep neural network for a simpler task may lead to poor performance of the neural network after training. This is because, training too many parameters than required will over-fit the neural network to get best performance on training dataset, once training is completed and test dataset is fed to the neural network, it does not perform very well. Similarly choosing a smaller and shallow neural network, may not be sufficient enough to extract the critical features required to perform the accurate classification.

AlexNet, GoogleLeNet, VGG Net, ResNet are some of the deep convolutional neural networks excelled at ImageNet Large Scale Visual Recognition Challenge (ILSVRC) in the past, which are trained to classifying a subset of ImageNet datasets into 1000 classes. But one of the main challenge in efficiently implementing the neural network is, these networks are very deep and require large number of parameters to define the network connections. The AlexNet uses 60 million parameters to classify the ImageNet with top 5 error rate of 15.3%. Similarly, VGG Net uses more than 130 million parameters on very deep convolutional network to classify images with top 5 error rate of 7.3%. Implementing such large network on modern processors with very limited cache memory or on-chip memory is challenging. In-order to improve performance on large neural networks, it is necessary to have the network parameters within the cache and avoid main memory accesses as much as possible. In other words efficient program reduce the number of off-chip memory. AlexNet trained with floating point weights will requires 340MBytes of memory to store the parameters, similarly VGG Net trained with floating point weights will requires 520MBytes

of memory. Such large networks cannot be able to fit into on-chip memory or caches, and require significant off-chip access to main memory every time run an inference. Network compression and pruning of neurons have been used to make larger neural networks fit into on-chip memory [14]. These methods exploit the inherent redundancy in the deep neural networks and remove the weaker connections to reduce the overall network size to fit into on-chip memory. During the training phase, Neural Network will learn which parameters or connections are important for the classification task, and the important connections will have significant value once trained.

Most of the times weak connections or insignificant parameters may not contribute to the final classification result. In many cases, those weak connections can be removed without having penalty in the prediction result. In classification tasks prediction result does not depend on the exact value of the output, rather depends on the probability value with respect to other classes. In the case of a neural network, classifying MNIST handwritten digits into 10 classes of digits from 0 to 9, final prediction digit depends on the argmax of the 10 class outputs or SoftMax of the output vector. So the network is resilient to small perturbations to the output value. Hence it does not produce wrong prediction if the probability of the digit predicted change from 1 to 0.95, after removing some insignificant parameters. So, the pruned network will end up in much smaller network parameters or smaller footprint of weight and a smaller number of computations, since we are not computing redundant connections. Another benefit of pruning is, it alleviates the over-fitting problem of trained neural networks. Since during pruning process less important connections are eliminated, the number of parameters defining the network are reduced, and the pruned network can be more generalized for the test data set.

In this work, we developed a framework that will reorder the individual operations in a classical feed forward network or multi-layer perceptron (MLP) by their significance towards the prediction result, and prune the insignificant computations to prevent executing insignificant computations. The major goal of this framework is to let the end-user to decide the amount of computation and memory footprint affordable for the inference, and make the network fit their resource budget while minimizing the accuracy loss. For example, an MLP network is trained to read license

plates. Suppose it requires 1 million FLOPs to detect a license plate image with 99% accuracy, and requires 4MB to store the network parameters. If only 10% of the operations are affordable, the target of the framework is to choose the right 0.1 million operations out of the 1 million operations available, while maintaining the prediction accuracy closer to the original network.

An extension of this concept is to do inference in an incremental fashion. Take the previous example, instead of executing all 1 million operations, start by executing sets of 10% significant operations at a time, until prediction outputs have a prediction confident in output prediction values. At the beginning, multiple classes can have similar probability, after executing few sets of operations the probability of the other will be reduced, and execution of buckets can be stopped once the required confidence level is reached. In the first example the number of computations for inference is fixed at 0.1 million, in the second example the number of computations is depended on the prediction confidence and varies from input to input.

True significance of an operation is a depended on the input to the network and the network parameter, and the input is only known at run-time of the algorithm execution. Calculating the significance of operations at run-time is not a feasible solution in most of the ML deployment situations and has to be computed at compile time. As the solution approximating the significance of operations can be evaluated at compile time, as shown in Section 2.3, and the approximate pruning and ordering of operations can be done based on the approximated significance.

2.1.2 Types of pruning

Pruning can be done at different levels depending on the type and size of the neural network. The larger the network is, the larger connections the network has, and typically more redundant connections to be pruned. Most of the execution time in classification networks are spent in convolutional layers and fully connected layers. The convolutional layer applies the convolution filter over a small portion of the input activation at a time, and produces multi-channeled output to be used by the following layer. The fully connected layer, as the name says, takes all the input activations from previous layer and produce output activations to be used by subsequent layers.

Whenever training a neural network for a specific task, the optimal network structure (number of layers, number of neurons in each layers) is typically unknown at training time. If the network structure chosen for training is larger than required the network will be forced to train redundant connections and over-fit for the training data.

Pruning of connections can be achieved at different levels in a neural network. It can be done by pruning a neuron and all the connections to/from the neuron (multiple connections at a time) or connection between two neurons (single connection at a time).

2.1.3 Multiple connections at a time

In a convolution operation, input activation and filter or kernel are element wise matrix multiplied to produce new output. Different filters used to generate different output channels from the same input. Usually the input image is larger than the filter size, so the filter is slid to different parts of the image to perform convolution. The purpose of the convolution operation is to search for specific features in the input image, which features to search is learned during the training phase. Together with multiple convolution layers, a convolution layer can be able to extract high level features from the previous convolution layers.

Choosing the filter size and the number of output channels is challenging. Large number of output channels may lead to extracting features that are not required for classification and small number of output channels may not be able to extract all the necessary features required for classification. Since convolution is compute intensive, pruning unnecessary channels can speed up the inference by avoid extracting unnecessary features.

Pruning multiple connection at a time is efficient in many convolutional layers. When a feature extracted by the convolutional layer from the input activations are less important for prediction, we can prune the particular feature extraction completely and eliminate the connections required for that feature extraction.

2.1.4 Single connection at a time

In fully connected layers, each output activation is connected to all the activations in the previous layer. The output is produced by a matrix multiply between the layer weights and input activation vector and followed by adding a bias vector to the result. Stronger the connection significant the weight values will be. Connections that are not strong have less significant impact on the network prediction. Those individual weak connections can be pruned to reduce the overall network size and the computation done without reducing the prediction quality significantly. Fully connected layers are used exclusively or together with convolution layers in many neural networks.

Unlike convolutional layers, there is no parameter reuse in fully connected layers for non-batch inputs. Since they only used once, minimizing the number of parameters required for the inference will minimize the FLOPs required and usually reduce the number of memory accesses.

2.2 Magnitude based pruning

This simple pruning method is used as a technique in machine learning to learn the important connections while training a large model. When training, the initial training phase focuses on learning the necessary weights or connections for the network to predict right, all the weights below certain threshold were ignored. After pruning the unnecessary weights, the sparse network is retrained with only the important connections to gain the accuracy back [13].

Below is the magnitude based algorithm used to prune the network connections.

2.2.1 Pruning algorithm

The following algorithm uses an intermediate representation of the neural network is stored in a CDAG format explained in Section 2.3.1.

Algorithm 1: magnitude based pruning

Input : CDAG,Percentage

Output: pruned.Network

thresholds[ ]=calculate.threshold(CDAG,Percentage)

foreach *layer in CDAG* **do**

  foreach *node in CDAG[layer]* **do**

    if *node.weight* < *threshold[layer]* **then**

      pruned.Network ← node

    end

  end

end

Algorithm 2: calculate threshold

Input : CDAG,Percentage

Output: threshold[ ]

order(node1,node2)=true: if | node1.weight| ≥ | node2.weight|

foreach *layer in CDAG* **do**

  newOrder=sort(CDAG[layer],order)

  value=newOrder[size(layer) × Percentage]

  threshold ← value

end

The algorithm takes the original neural network and the pruning percentage, and for each percentage (p%) of pruning given to the program, first calculate the threshold value of weights in each layer, from which any weights below will be pruned such that the remaining network will be containing %p of the original network.

It prunes nodes in a layer independently from the other layers, it is useful in the early stages of training to figure out which connections in each layer learns important information and focus on training those important connections from thereafter. By learning only important connections, the iterations to train accurate model will be reduced and the trained network will be less likely to suffer from over-fitting.

Our experiments show, in fully trained network without retraining, magnitude based pruning suffers from significant accuracy loss. So, an efficient pruning for deployment neural networks must consider the significance of the operation to the final accuracy than just the magnitude of the weights.

2.3 Significant based pruning

Each node or operation in the inference program has a significant value towards final prediction. Significance of each node in the CDAG is depended on the weight and the input to that node. Approaches that consider only the weights will not identify the significant operations in the CDAG. In order to determine the significance of the operations, the input and the weight value are required. Consider the following example program.

$$y = y + w_1 * x_1$$

$$y = y + w_2 * x_2$$

Figure 2.1: example program

In the above example, the first operation computes the value of y which is depended on the weight w_1 and input value x_1 and the previous value of y. Similarly the second operation computes the value of y from weight w_2 and input x_2 and the previous value of y. If we want to prioritize the significant computation for the output y we need to compare the significance of both operations.

In order to compare the significance of the operations above, we need the values of w_1,w_2, x_1,x_2. We have the value of the weights in a trained network, but the input values x_1, x_2 are known at run-time only. Calculating significance at run time is not a feasible solution, significance need to be estimated at compile time before deployment. In order to evaluate the significance at compile time, we are estimating the approximate significance of the operations. In other words, we have

to estimate the approximate value of the inputs (x_1,x_2) to the operations, which is explained in Section 2.3.2.

2.3.1 Intermediate Representation of the program

In order to process the neural network for significant based pruning, a CDAG (Computational Directed Acyclic Graph) based internal representation is used to store the original inference program. CDAG is used to capture every computation executed in the inference program.

input program trace

The run-time trace of the inference program is used to extract the computations and the weight values used in the inference program. The inference program is augmented with the printf calls to output the run-time trace containing computations of the inference program and the weight values associated with it. Fig.2.2 shows run-time trace of an inference program. Each line in the run-time trace corresponds to a unique operation in the inference program. Each trace line contains the following fields, unique id corresponding to the operation, iteration vector of the operation, memory cells accessed by the operands. Unique id expresses the original schedule of the program, iteration vector specifies the loop iterator's values including the layer number and memory cell accessed is used to capture the dependencies in the inference program.

```
ID:777 DIM=3 rd=0.310900 rs1=0.310900 rs2=-0.254460 rs3=0.000000 l=1 i=0 j=777 &rs1=0x7ffd5217e9f8 &rs2=0x7ffd52182838 &rs3=0x7ffd5217e9b8 &rd=0x7ffd5217e9f8
ID:778 DIM=3 rd=0.310900 rs1=0.310900 rs2=-0.358700 rs3=0.000000 l=1 i=0 j=778 &rs1=0x7ffd5217e9f8 &rs2=0x7ffd52182840 &rs3=0x7ffd5217e9c0 &rd=0x7ffd5217e9f8
ID:779 DIM=3 rd=0.310900 rs1=0.310900 rs2=-0.365590 rs3=0.000000 l=1 i=0 j=779 &rs1=0x7ffd5217e9f8 &rs2=0x7ffd52182848 &rs3=0x7ffd5217e9c8 &rd=0x7ffd5217e9f8
ID:780 DIM=3 rd=0.310900 rs1=0.310900 rs2=-0.297770 rs3=0.000000 l=1 i=0 j=780 &rs1=0x7ffd5217e9f8 &rs2=0x7ffd52182850 &rs3=0x7ffd5217e9d0 &rd=0x7ffd5217e9f8
ID:781 DIM=3 rd=0.310900 rs1=0.310900 rs2=0.004840 rs3=0.000000 l=1 i=0 j=781 &rs1=0x7ffd5217e9f8 &rs2=0x7ffd52182858 &rs3=0x7ffd5217e9d8 &rd=0x7ffd5217e9f8
ID:782 DIM=3 rd=0.310900 rs1=0.310900 rs2=0.006153 rs3=0.000000 l=1 i=0 j=782 &rs1=0x7ffd5217e9f8 &rs2=0x7ffd52182860 &rs3=0x7ffd5217e9e0 &rd=0x7ffd5217e9f8
ID:783 DIM=3 rd=0.310900 rs1=0.310900 rs2=0.005578 rs3=0.000000 l=1 i=0 j=783 &rs1=0x7ffd5217e9f8 &rs2=0x7ffd52182868 &rs3=0x7ffd5217e9e8 &rd=0x7ffd5217e9f8
ID:784 DIM=3 rd=0.310900 rs1=0.310900 rs2=-0.020006 rs3=0.000000 l=1 i=0 j=784 &rs1=0x7ffd5217e9f8 &rs2=0x7ffd52182870 &rs3=0x7ffd5217e9f0 &rd=0x7ffd5217e9f8
NON_LIN 0x7ffd5217e9f8
```

Figure 2.2: example program trace

CDAG representation

CDAG is a 4 tuple of (I, V, E, O), V are the vertices of the graph, I is an input set with no incoming edges, E is the set of edges between vertices, O is the output set of the graph. $V \backslash (O +$

I) represents the computation vertices. Each line in the run-time trace will be represented by a computation vertex and the edges between vertices represents the flow of operands in to the computational vertex. The CDAG representation enables a more detailed program representation for fine grain optimization.

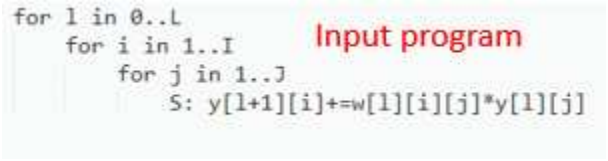


Figure 2.3: example program

The program in Figure 2.3 represents a Multi-Layer Perceptron(MLP) network. The iteration vector $< l, i, j >$ represents the domain of the statement S. The address of variables $y[l][j]$ and $y[l+1][i]$ are captured in the trace in order to construct the dependencies in the program. The following notation is used in the explanation below, iteration vector $< l, i, j >$ represents iteration space of the of the MLP statement S. Lower case letters l, i, j represents variable parameters of the iteration vector and upper case letters L, I, J represents constant/one instance of l, i, j respectively.

Two important dependency patterns are expressed in the program, First Read After Write (RAW) dependency of y, where $< l-1, X, j >$ write to $y[l][X]$ and $< l, i, X >$ read the value $y[l][X]$, both accessing the same data. In neural network context, each consecutive layer l read activations from outputs written by previous layer $l-1$. So the final schedule must respect this dependency for correct program execution.

The second dependency pattern is Write After Write (WAW) and RAW found in the reduction operation within a layer, where two $< L, I, j_1 >$ and $< L, I, j_2 >$ will read and write to $y[l][i]$. But if we assume associative reordering is valid, we can eliminate the dependency of the reduction. In this work we assume associative reordering is legal, and present a new schedule that is based on significant computation first.

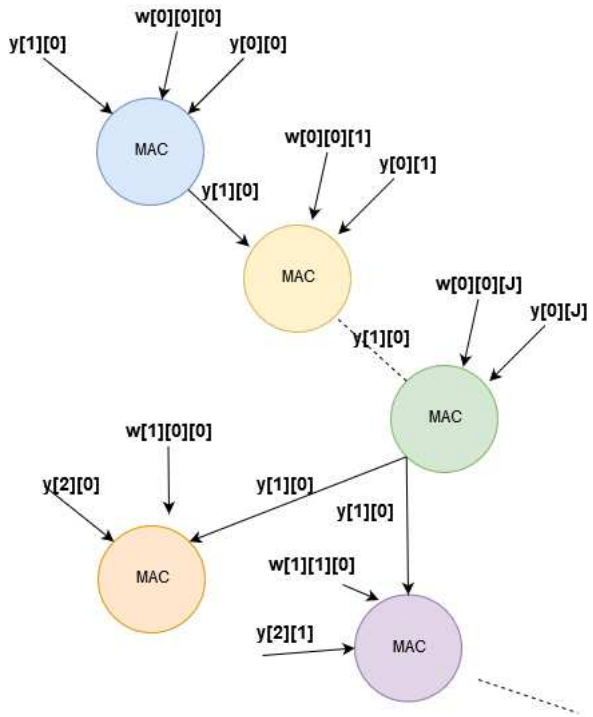


Figure 2.4: CDAG representation of example program in 2.3

Figure 2.4 shows the CDAG constructed from the trace in Figure 2.2. For simplicity multiply operator and the addition operator in the statement S in Figure 2.3 are fused together and represented as a single multiply-accumulate (MAC) operator. So each line in the trace or each instance of statement S is mapped to unique vertex in the graph. In Figure 2.4 each vertex is marked by different color and the dependency pattern constructed from the memory accesses are represented by the edges in the graph. Each MAC node in the graph has 3 incoming edges for the flow of input x, weight w and previous value of the output y; and one output edge for the flow of computed value $y + w * x$. The dependency patterns are easily observable in the CDAG. x does not has to be the input image for layers other than 1st layer. If we consider the input edges to a vertex, there is no dependencies for the weight values since they are read only values, but input edge x may have a dependency if it has been written by any of the previous nodes, and input y has a dependency from the previous node's write operation or output edge.

2.3.2 Contribution Table

As we seen in the example 2.1, in order to identify which operations are significant for the neural network, we need to estimate the input values to the operations at compile time. The objective of the contribution table is to estimate the approximate impact of each input pixel to the final outputs of the classification network. In classification networks, the last layer is usually a SoftMax layer. The SoftMax layer transform the output from the previous layer to a probability between 0 and 1.

$$\sigma(y)_i = \frac{e^{y_1}}{\sum_{j=1}^{N} e^{y_j}} i = 1 \dots N \tag{2.1}$$

Eq. 2.1 represents the SoftMax layer, where N is the number of classes in the output layer, y_j is the j^{th} class output. For the significance analysis, we can neglect the SoftMax layer and work on the outputs before SoftMax y_j $j = 1 \dots N$. So the significant table must represent the approximated impact of each input pixels on each output y_j. Using the contribution table, we can identify which part of the input image are important to classify the image as class 1, class 2 ... class N for a given trained network.

Below is the algorithm to estimate the contribution of the inputs. It generates a table containing approximated significance of the input to each output class. The table generated has M rows and N columns, where N is the number of output classes and M is the number of inputs. i^{th} row and j^{th} element in the table represents the impact of i^{th} image pixel towards j^{th} class. For example, in a neural network trained to detect handwritten digits, the neural network will be learned to neglect the dark pixels in the four corners of the image, since they contain no information. So, the contribution table has lower values for those pixels. More interesting example would be, consider the digit zero in the handwritten digit dataset. It usually contains dark pixels in the center of the image, So the contribution table will usually contain lower or negative contribution for those pixels in the table for output class zero. An important note is, the pixel table is constructed from the given trained network, so the contributions may vary between two networks because of the way they trained. The contribution table captures the simplified Saliency map [15] of the network.

Algorithm 3: contribution table

Input : CDAG

Output: Table[ ] [ ]

foreach *pixel in pixels* **do**

  foreach *output in CDAG(O)* **do**

    Table[pixel][output]=output value(forward pass(CDAG,pixel),output);

  end

end

CDAG(O) represents the output nodes in the CDAG, and CDAG(I) represents input nodes to the graph, which are individual pixels in the input image. The contribution is evaluated by activating pixels and the measuring the output values after a forward pass through the graph. Each contribution is measured at by activating individual pixels in the image, and the linear relationship between the input and the output are captured in the contribution table. Approximating the network with linear function enables to eliminate contribution saturation.

$forwaardpass()$ function does the forward sweep of in the CDAG with the given activated input pixels. In this case, it can be seen as an image with single pixel active, fed in to inference network. $outputvalue()$ function returns the output node's final computation value. $ToArray()$ function maps the nodes in the graph to an index accessible array of nodes.

Algorithm 4: forward pass

Input : CDAG,pixels

Output: CDAG

foreach *pixel in CDAG(I).pixel* **do**

    if *pixel* $\subset$ *pixels* **then**
        ToArray(CDAG(I))[pixel].Y=ACTIVE
    end
    else
        ToArray(CDAG(I))[pixel].Y=REFERANCE
    end

end

foreach *n in CDAG(V \(I+O)* **do**

    n.X=ToArray(CDAG(V+I))[n.pred_x].Y
    n.Y=n.Y + n.W * n.X
    n.Significance=n.W * n.X

end

Each pixel in the input images is being activated one pixel at a time and the output class's impact is recorded in the table, every other pixel is set to REFERENCE value of the pixel for the contribution table estimation. The values for the ACTIVE and REFERENCE pixels are taken from the training dataset. ACTIVE pixel value is the maximum value of the pixel when pixel is on in the input image, and REFERENCE value of the pixel when the pixel is off in the input image.

Algorithm 5: output value

Input : CDAG,output

Output: Y

foreach *node in CDAG(V)* **do**

    if *ToArray(CDAG(O))[output].addr==node.rd_addr* **then**
        Y=node.rd;

end

Relationship between input activation and output class

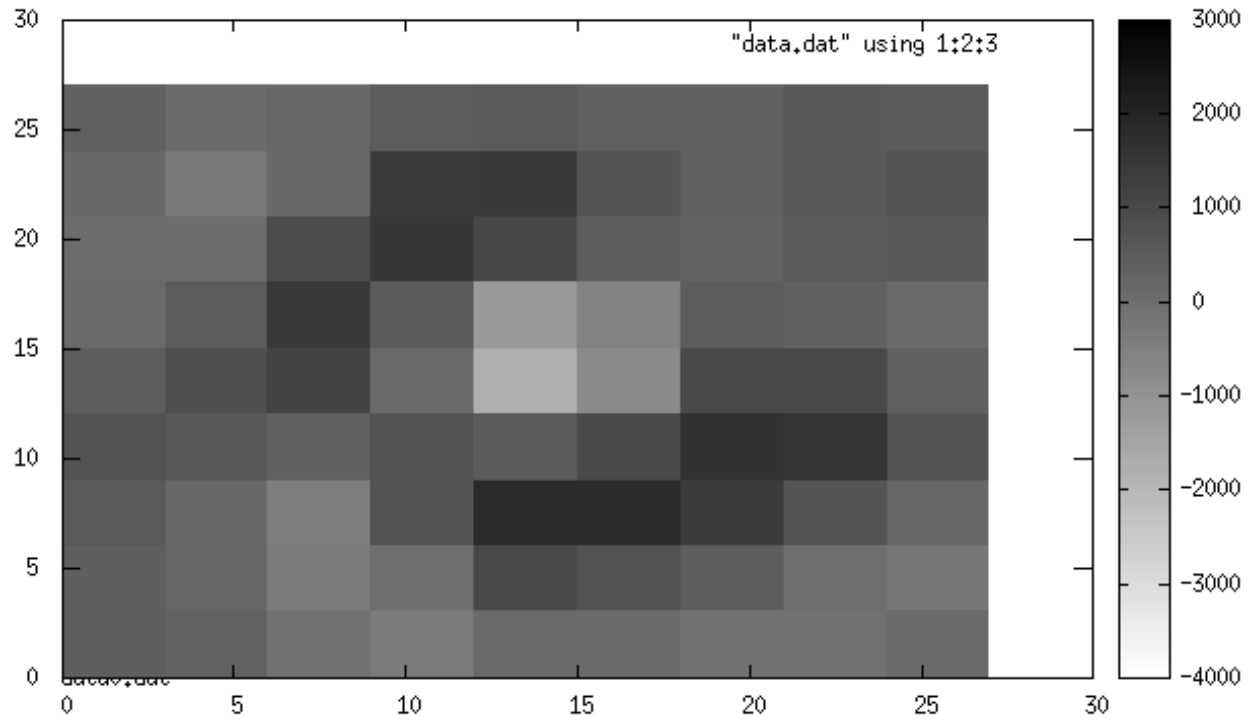


Figure 2.5: Heatmap representation of input contribution for Class 0 of MNIST classification network

Fig. 2.5 shows the heat-map representation of an input contribution for output class 0 generated from a 5 layer fully connected inference network to classify MNIST handwritten digits. The network is trained with MNIST data set to classify the centered input image 28px $\times$ 28px into 10 different classes ranging from digit 0 to digit 9.

The above figure is the contribution table generated for the output class 0 to classify digit zero. The input pixels represented as dark in the figure represent that, those pixels will increase the contribution of class 0 if they present in the actual input image, similarly the lighter pixels in the figure represent that, those pixels will reduce the contribution to predict class 0 if they present in the actual input image. The pixels near the edges have lower contributions and less important when predicting the digit.

Contribution table will assist us to choose the input pixels that are more important to the classification of image. Significant input activations from the contribution table can be used to ap-

proximately estimate the run-time input activations. The contribution table used to generate an approximated input image activation for maximum activation of each classes. The significance calculated serves as an approximated significance of nodes for each class in the network. Without approximated contribution table, there will be intractable number of possible significances for each node, depending on the run-time input present.

2.3.3 Reordering of nodes in the CDAG based on significance

The purpose of the reordering of nodes in the CDAG is to generate an approximated order of operations based on significance such that most significant operations executed earlier in the program. In an event of early program termination, get partial inference prediction results as close as possible to the full inference.

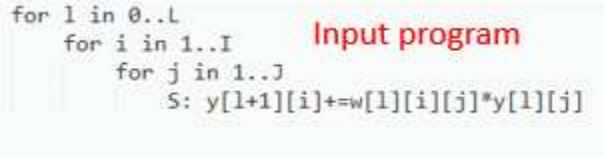


Figure 2.6: example program

We revisit the example program in Fig. 2.3 and consider reordering the computations in the inference program based on significant. The goal is to find a different legal schedule that respect that significant operations ordered first. A legal schedule must respect the dependencies on the original program, but we are relaxing the dependency of the reduction operation, assuming floating point associative reordering is valid. This assumption gives us the flexibility to reorder operations based on significance.

Consider the program instance $< l = I, i = I, j = J >$, the significance of an operation in this case is the value computed by $w[L][I][J] \times y[L][J]$. Calculating the significance requires the value of $w[L][I][J]$, which is already known, and $y[L][J]$. The value of $y[L][J]$ depends on the previous computations that write to it, and those computations led to it. In other words, the significance of operation at instance $< L, I, J >$ is depended on a sub-graph G_1 of nodes of the original CDAG,

which can be determined by backtracking from the node instance $< L, I, J >$. Estimating the value of the $y[L][J]$ requires populating the contribution from the sub-graph that writes to $y[L][J]$.

Using the contribution table, mock image IMG_1 with only significant pixels can be activated in the input, which approximates the significant pixels in the input image for the actual classification at run-time. Now the approximated value of $y[L][L]$ can be calculated by computing the sub-graph $G_1(IMG_1)$ for the approximated input IMG_1. So the significance of operation at instance $< L, I, J >$ can be calculated as $w[L][I][J] \times G_1(IMG_1)$.

Each operation has different significance towards different classes. Some operations are more significant for some classes than others. So reordering of operations based on significance has to represent fair number of operations for each class prediction. Otherwise the reordered schedule may not be able to predict all the classes in the network accurately.

Equal share of operations

In order to implement fair reordering, the operations are not reordered individually, instead operations are grouped in to buckets and reordering is done based on buckets. Buckets contains roughly equal number of operations reordered based on significance to each output class. For example, consider a program with parameters NL=10, NI=100, NJ=1000, and output classes of 100. Total number of operations in the input program is 1 Million. Assume we are reordering the total number of operations into 10 buckets, so each bucket roughly contains 100,000 operations. Each of the operations inside a bucket will be split equally to represent significance operations for the 100 classes. So executing only the 1st bucket from the 10 buckets will lead to executing the significant 10% operations of the inference program.

In order to generate the mock image IMG_1 with significant pixels, a ranking order of pixels for each class is generated from the contribution table as shown in Algorithm 6. If there are 10 output classes in the network, there will be 10 different ordering of pixels generated.

Algorithm 6: rank_pixels

Input : Table[ ][ ]

Output: RankedPixel[ ][ ]

order(pixel1,pixel2)=true: if $| Table[pixel1][x] | \geq | Table[pixel2][x] |$

foreach *x in CDAG(O).size* **do**
| RankedPixel[:][x]=quick_sort(Table[:][x],order);

end

Since these ranked pixels are generated from the contribution table, the rank represents how important these pixels are in the input image to do the classification right. For example, pixels with higher rank for class 0 will contribute more towards the class 0 classification.

Once the pixels are ranked, significant operations can be approximated by passing the approximate input image with significant pixels to the inference CDAG. The Algorithm 7 shows the reordering operations in to set of operations based on significance. The number of sets to execute are given by the user, and for now assume there are 10 sets of operations to reorder a classification network with 10 classes. The algorithm first identifies the operations for the 1^st set, that is the most significant 10% operations in the CDAG. In order to find the significance of operations in the CDAG, the algorithm activates the most contributing 10% pixels of the input image for each class from 1 . . . NClasses. Since we are using the qual share of operations across classes, each set should contain equal number of operations towards each of the classes. So the initial significant 10% set will have significant 1% operations for each of the 10 classes. Note that percentages are based on the total number of operations in the CDAG.

Algorithm 7: Reordering

```
Input  : CDAG,Table, NSets
Output: Sets{ }
Ranked_Pixels[ ][ ]=rank_pixels(Table[ ][ ])
NClasses = size(CDAG(O))
 NInputs = size(CDAG(I))
 foreach  i in 1 ... NSets do
    foreach j in NClasses do
       Sig_pixels{} ← Ranked_Pixels[1 .. NInputs * i /NSets][j]
       CDAG=forwardpass(CDAG,Sig_pixels)
       nodes={ n⊂ Nodes(CDAG(V)):n.rd_addr=ToArray(CDAG(O))[j].addr}
       while nodes ≠ ∅ do
          nodes=select_subgraph(sort(nodes),NSets)
           Sets[i]+=nodes
           CDAG = CDAG \ nodes
           nodes=predecessor_nodes(CDAG,nodes)
       end
    end
end
```

Significant operations for each set and output class are evaluated by a forward pass of estimated significant inputs and backward tracking of operations through the CDAG. The forward pass will help to estimate the significance of operations in the CDAG for the given Sig_pixels and the backtracking process will select a sub-graph of operations ($select\_subgraph$) that are significant at each stage of backtracking. Continuing our previous example case, the backtracking will return a sub-graph with 1% of the total operations. Fig. 2.7 shows the visualization of the backtracking process.

Algorithm 8: predecessor_nodes

Input : CDAG,nodes

Output: predecessor_nodes{ }

foreach *node in nodes* **do**
| predecessor_nodes+={ n ⊂ CDAG(V):node.predecessor_rs3==n}

end

Algorithm 9: sort

Input : nodes{ }

Output: ordered_nodes{ }

order(node1,node2)= true: if $\mid node1.Significance \mid \geq \mid node1.Significance \mid$

return ordered_nodes=quick_sort(nodes,order)

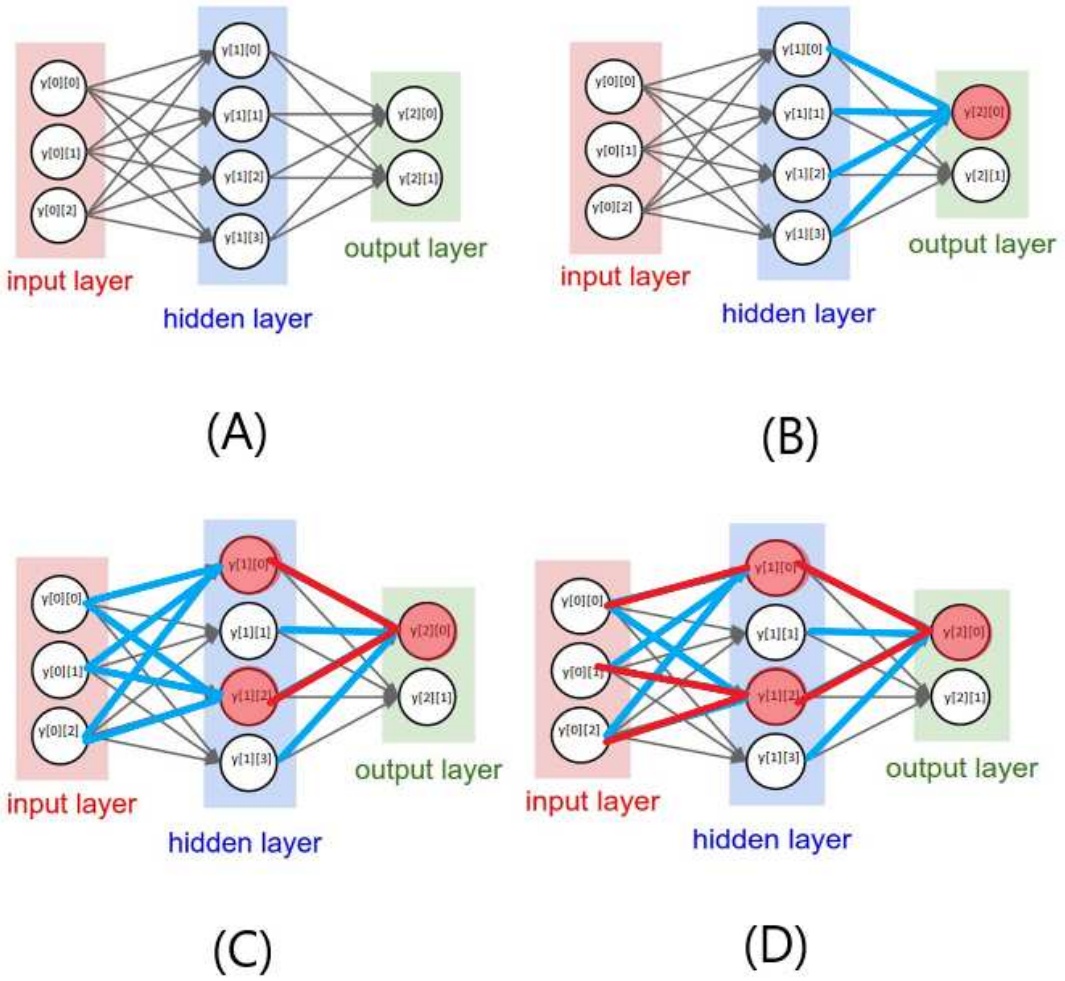


Figure 2.7: Backtracking process

Fig.2.7 (A) is the network state after the forward pass with estimated significant input, (B) shows the backtracking starting with the connections required for the first output class, (C) selecting the connections with higher significance, (D) redo the step (B) with the new neurons found in step (C) until reach the input neurons. Then follow the same procedure for the next output class. The procedure removes the selected connections from the network and iteratively select the next significant connections.

Chapter 3

Codelets aware pruning

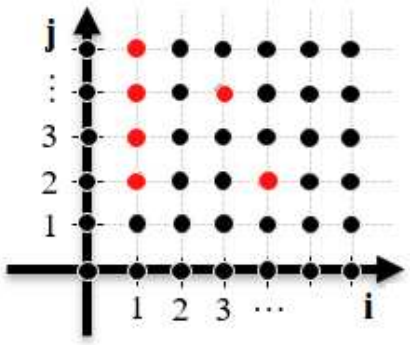
3.1 Overview

3.1.1 Purpose

Reordering based on significance of the operations will provide a schedule that will ensure that executing a sub-graph of operations to achieve good prediction. But executing sub-graph of operations does not necessarily produce better performance, compared to executing the original inference graph. Modern computers support parallel execution of operations using vector units, and modern compilers take advantage of this vector units by SIMD vectorizing the input program when available.

The original input neural network before pruning, can be easily transformed into nicely vectorized program using a simple loop permutation. Since a pruned network skips insignificant operations, the program schedule of pruned network has a highly sparse computation pattern.

Due to the sparse structure, the final network will be represented in a sparse structure, which in turn result in a sparse matrix vector program. Traditional Sparse Matrix Vector (SPMV) programs stores sparse matrix in Compressed Sparse Row (CSR) or Compressed Sparse Column (CSC) for efficient storage by storing only the non-zero elements of the matrix. Which results in complex sparse matrix vector product program with indirect memory access functions. Which prevents any advanced optimizations the compiler tries to do.



(A)

```
for(i=0; i< N; i++)
    for(j=0; j<M; j++)
        y[i]+=Matrix[i][j]*x[j];
```

(B)

Figure 3.1: (A) Sparse Matrix, (B) Matrix Vector program

The Fig. 3.1(A) represents a N $\times$ M matrix, where the non-zero elements in the sparse matrix are represented by red dots and zeros in the matrix are represented by black dots. The program on (B) is a generic matrix vector product iterate over N $\times$ M iteration space. Since the matrix is largely consist of zero elements, the generic matrix vector program executes useless computations most of the time. Sparse matrix vector programs overcome this problem by only iterating over the non-zero elements of the matrix. The operations multiplied with zeros are skipped in sparse computation. Compressed Sparse Row (CSR) program for the above sparse matrix computation is shown in Fig. 3.2 below.

```
for (i = 0; i < nrows; i++)
    for (j = pos[i]; j <= pos[i+1]; j++)
        y[i] += csr_data[j] * x[cols[j]];
```

Figure 3.2: CSR program

The CSR representation stores only the non-zero elements of the matrix and their positions. In order to determine the iteration space, the program has to access the stored non-zero element's position. For each iteration, the it needs to read 4 memory locations, and the loop bounds are

determined by the number of non-zeros in a row. Even though the program is now computing only on the non-zero elements of the matrix, the CSR program is very difficult to analyze. Since the program is generic for any sparse matrix, the loop bounds are depended on the input sparse matrix and the indirect memory access patterns, the program vectorizability cannot be determined at the compile time and makes it difficult for the compiler to apply any advanced optimizations to the program.

The paper on Generating Piecewise-Regular Code from Irregular Structures [16] presented a new methodology to generate SPMV programs without accessing any indirect arrays. This method works by, instead of having a single nested loop accessing the non-zero memory locations in the original program order, generates a new schedule which comprise of one or more nested loops accessing the non-zero memory locations. In order to eliminate the indirect array, the approach partition the iteration space into multiple polyhedra, which can be later represented by affine polyhedral expressions. A single polyhedron will represent one or more iteration points in the SPMV. It eliminates the need for indirect arrays to represent loop bounds, and generates affine loop iterators, which can be easily processed by the optimizing compiler.

```
for (j = 2; j <= 5; j++)
    y[1] += csr_data[j-2] * x[j];
y[3] += csr_data[5] * x[4];
y[4] += csr_data[6] * x[2];
```

Figure 3.3: Piecewise-Regular program for the sparse computation in Fig. 3.1

Fig. 3.3 shows the new sparse computation generated from piecewise regular structures. The 4 consecutive non-zero computation iterations with i=1 are grouped into a single loop representing a regular polyhedron, and the remaining 2 no-zero computation iterations represents the leftover points which cannot be grouped in to any another polyhedra. The loop bounds of the program are

known to the compiler and the indirect memory access are eliminated, hence the program can be easily analyzed by the compiler.

Since this methodology tries to find polyhedra from the sparse matrix, the output program is depending on the input sparse structure that was fed. Having good structured input sparse matrix will minimize the code size and simplify the polyhedra representation. The performance of the output program depended on the number of vectorizable operations, and the code size. The less number of simple polyherda the lower the output code size will be. So the performance will improve if most of the operations can be represented by vectorizable polyhera.

Since the schedule from significance based pruning has no control over the sparsity structure, the performance of the pruned inference is difficult to characterize. Codelets aware reordering will enable to control the structured sparsity within the significance based pruned networks, by making minimal changes to the pruned network to maximize the structured sparsity.

3.2 Codelets

The codelet is a lattice shape to represent iterations that can be represented by a polyhedron. Codelet aware reordering will find codelets within the schedule to improve performance. More the number of codelets, better the output implementation will be.

3.2.1 Codelet shapes

Codelet shape is represented by the number of iterations in each dimension, and the stride between the iterations along each dimension. In the Fig. 3.1(A), the iteration points presented by the polyhedron D_1={[i,j]: i=1 $\wedge$ 2<= j <=5} can be extracted from the codelet shape of <1,4,1,1>. Where the first two numbers represent the number of iteration points along each dimension i,j; and the next two numbers represent the stride between each iteration along each dimensions.

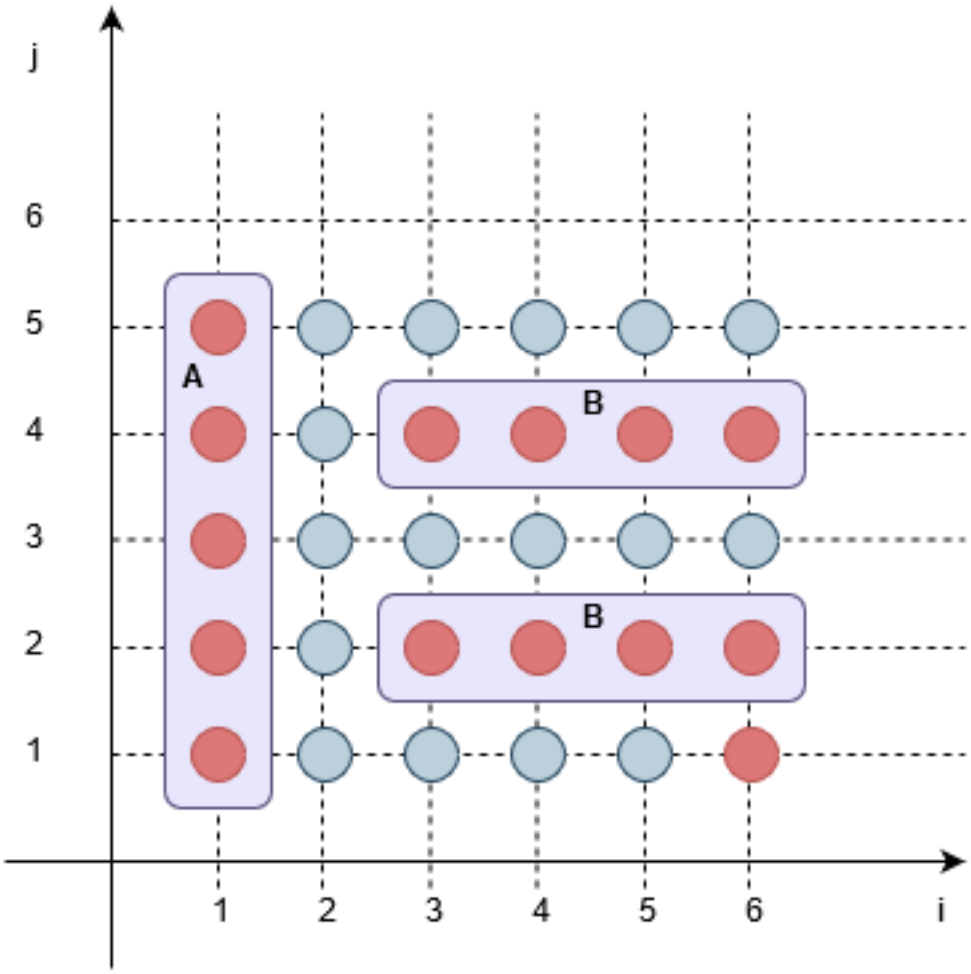


Figure 3.4: Sparse iteration space with regular structures

Fig.3.4 shows a 2-dimensional space with sparse iterations with non-zero weights marked in red points. The colored areas A and B represents 2 codelet found in the sparse iteration space. A represents a vector of 5 iterations along j dimension with codelet shape of <1,5,1,1>, B represents a 4×2 vector with stride of 2 along j dimension with codelet shape of <4,2,1,2>.

3.2.2 Codelets aware reordering

The original reordering based on significance of operations generates a SPMV program, In order to improve the SIMD performance of the significant based reordered schedule, we can make small changes to the schedule such that it will generate better structured SPMV with better vec-

torizability. When executing the reordered program, the user will specify the number of sets to execute. If the user specifies 30% operations to be executed, the operations are divided in to 10 sets by significance and the first 3 sets out of 10 will be executed. Since the significant based schedule is executed by sets of operations, reordering operations within a set is allowed and will not change the prediction accuracy. When reordering operations across sets, it can change the prediction accuracy. For example, in the previous example when moving an operation from 4^{th} set in to 3^{rd} set, the new schedule will have a less significant operation from 4^{th} set and a significant operation from 3^{rd} set will be pushed out from the executed schedule.

Reordering based on significance of the operations only will result in a total order that may have highly irregular sparse computation. In order to optimize the final trace for better performance, implementation of efficient codelet aware reordering of operations in the final total order in necessary. A threshold of maximum allowed perturbations or edits to the reordered operations is set, in order to limit the loss after reordering.

Algorithm 10 is used to generate a new schedule which is aware of codelet in the final reordering. For a given reordered operations based on the significant contribution, the algorithm perturbs the original ordering, allowing operations to move from the less significant set to significant set in order to maximize the number of codelets in the final reordering. A perturbation threshold (edit distance) is applied to control deviation from the original significant based reordering.

Each edit moves one operation up in the significance based reordered set of operations, where top operations are more significant than the bottom operations. Fig. 3.5 shows the codelet aware reordering trading off between structured sparsity and significance based total ordering.

Algorithm 10: codelets finding

Input : nodes{ },Shapes{ }, $MAX\_EDIT\_DISTANCE$

Output: reordered_nodes{ }

N=nodes.size()

totalEditDistance=0

for *i in 1 .. N* **do**

  for *S in Shapes* **do**

    t{ }←nodes.toArray([i:i+1+$MAX\_EDIT\_DISTANCE$])

    codelet=findShape(S,$node[i]$,t)

    if *(codelet!=NULL)* **then**

      e=editDistance(codelet,node[i])

      if *e+totalEditDistance*<=$MAX\_EDIT\_DISTANCE$ **then**

        insert(codelet,nodes)

        totalEditDistance+=e

  end

end

return nodes

It takes the previously generated schedule based on significant as input and a file containing codelet shapes to explore.

Algorithm 11: findShape

Input : Shape S,node1,nodes{ }

Output: nodes{ }

t{ }← node1

foreach *n in nodes\ node1* **do**

  if *validMember(n,node1,S)* **then**

    t{ }← n;

  code=getValidCodelet(t,S) **if** *code!=NULL* **then**

    return t

end

Algorithm 12: validMember

Input : new_node,node1,S

Output: bool

foreach *d in S.dimension* **do**
  if |*new_node.index(d)-node1.index(d)*|>*S.length(d)*× *S.stride(d) or*
  (new_node.index(d)-node1.index(d)) $(\mathrm{mod}\ )$ *S.stride(d) !=0* **then**
    return false;

end

return true

Algorithm 13: getValidCodelet

Input : nodes{ },Shape S

Output: nodes{ }

t{ } ← LexicographicSort(nodes)

code={ }

foreach *n in nodes* **do**
  code ← n
  foreach *m in nodes**n* **do**
    if *(m.index(d)-n.index(d))* <*S.length(d)*×*S.stride(d) :* $\forall d \in S.dimension$ **then**
      code ← m
  end
  if *code.size()==Shape.size()* **then**
    return code
  else code={ }

end

Algorithm 14: editDistance

Input : codelet{ },baseNode

Output: distance

d=0

foreach *n in codelets* **do**
  d+=n.rank-baseNode.rank

end

return rank

Algorithm 15: LexicographicSort

Input : nodes{ }

Output: ordered_nodes{ }

order(node1,node2)= true: if node1.index $\prec$ node1.index

return ordered_nodes=quick_sort(nodes,order)

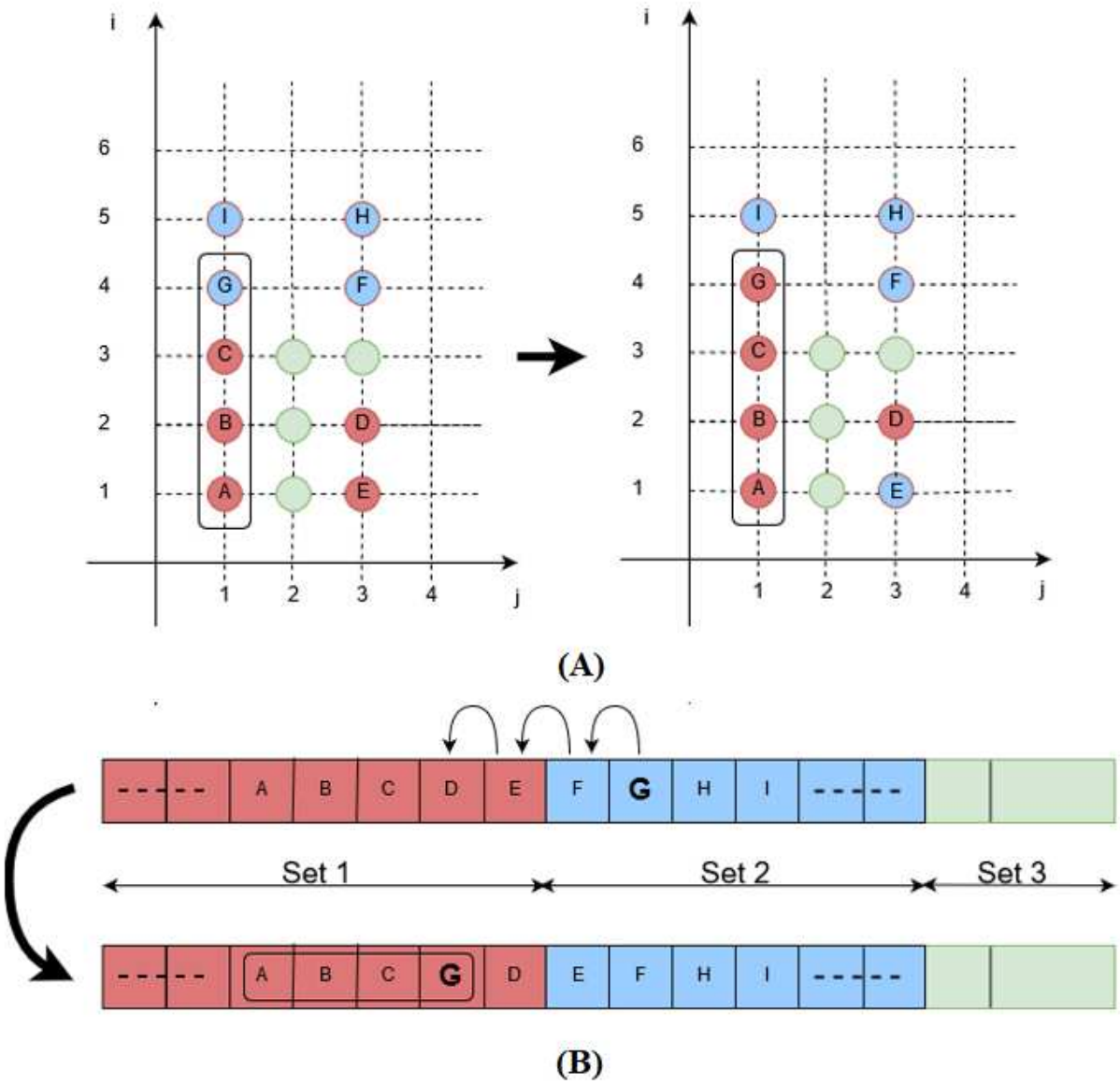


Figure 3.5: Codelet aware reordering

In the above Fig. 3.5, the plot (A) shows a single layer network, and the iteration space of sparse computations in 2-D space before and after codelet aware reordering. Ordering (B) represents the total order of the operations before and after codelet aware reordering the same network. Operations in red represents the 1^{st} significant set, operations in blue represents the 2^{nd} significant set and operations in green represents the 3^{rd} significant set. Assume we only execute the 1^{st} set, then the iteration space will only have A,B,C,D,E points in the 2-D space. Now consider moving

the operation G from 2^{nd} set to 1^{st} set. The new ordering is now has a codelet of <4,1,1,1>, but the operation E has to be moved from 1^{st} set to 2^{nd} because each set must have equal number of operations, which implies the the new 1^{st} set now traded off a significant computation to have a codelet shape. The codelet aware reordering will trade off between significance based schedule and efficient codelet based schedule based on the threshold provided. Moving is allowed between next significant set only and moving operations across multiple sets is prohibited to prevent huge accuracy drops.

Chapter 4

Evaluation of Network Pruning

4.1 Experimental setup

The prediction quality results showed in this section were from the neural networks to classify the MNIST handwritten digits [17]. The MNIST dataset contains 60,000 images and labels of handwritten digit images for training and 10,000 images and labels for testing. Each digit is centered and image is size normalized to 28px $\times$ 28px single channel gray-scale image as shown in Fig. 4.1. The label value is between 0 to 9.

Figure 4.1: Example collection of images from MNIST database[2]

The Classical Feed Forward Neural Networks are used in the experiment to classify the MNIST images into 10 classes. Classical Feed Forward Neural Networks with different number of layers with varied number of hidden units are trained with PyTorch framework [18]. The input image is

standardized before training. Mean and standard deviation of the training data set are calculated for each image pixel, and the standardized input is used in training and testing of the network as shown in Algorithm 16. The prediction accuracy is evaluated after pruning to 33%, 50% and 67% of connections of the original networks or 3X, 2X and 1.5X compression of the network respectively.

Algorithm 16: Standardization

Input : image[ ], mean[ ], std[ ]

Output: InputImage[ ]

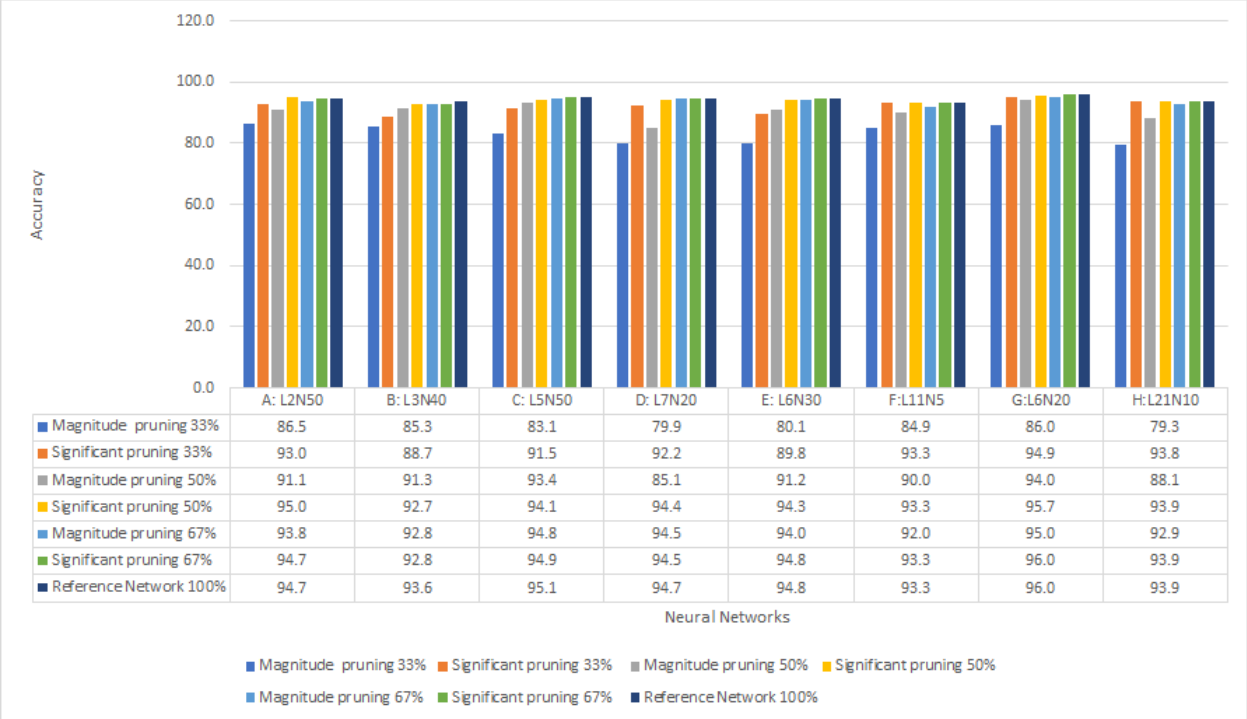
foreach *pixel in image[ ]* **do**
| InputImage[pixel]=(image[pixel]-mean[pixel])/std[pixel];

end

return InputImage[ ]

4.2 Magnitude based pruning accuracy

The network connections are pruned based on the algorithm in Section 2.2, where the connections below certain threshold are pruned. Fig. 4.2 compares the accuracy of the network after pruned based on magnitude and pruned based on the significance Sec. 2.3. The reference network is the original network without pruning any connections. The L2N50 represents a trained network with 2 hidden layers and 50 hidden units within a layer.



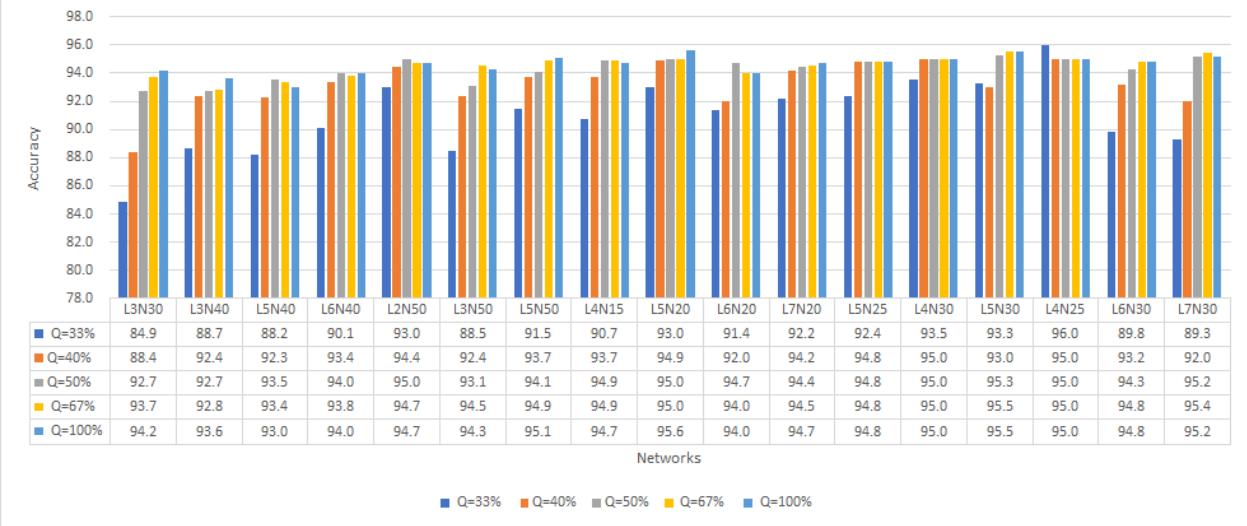
	A: L2N50	B: L3N40	C: L5N50	D: L7N20	E: L6N30	F:L11N5	G:L6N20	H:L21N10
Magnitude pruning 33%	86.5	85.3	83.1	79.9	80.1	84.9	86.0	79.3
Significant pruning 33%	93.0	88.7	91.5	92.2	89.8	93.3	94.9	93.8
Magnitude pruning 50%	91.1	91.3	93.4	85.1	91.2	90.0	94.0	88.1
Significant pruning 50%	95.0	92.7	94.1	94.4	94.3	93.3	95.7	93.9
Magnitude pruning 67%	93.8	92.8	94.8	94.5	94.0	92.0	95.0	92.9
Significant pruning 67%	94.7	92.8	94.9	94.5	94.8	93.3	96.0	93.9
Reference Network 100%	94.7	93.6	95.1	94.7	94.8	93.3	96.0	93.9

Figure 4.2: Prediction accuracy comparison between magnitude based pruning and CDAG based insignificant pruning

The Neural Networks A-E represents fully connected classical neural networks; F, G and H represents cascade neural networks. When 3X compression applied, the accuracy of the networks pruned based on magnitude only is on average 9.3% below the reference network. The Network D with 7 layers with 20 hidden units after 3X compression (33% of original connections remaining) shows 79.9% accuracy on the test dataset, which is 14.8% accuracy loss compared to the original inference accuracy of 94.7%. Whereas 3X compression after significance based pruning shows

92.2% accuracy with only 2.5% accuracy loss. Once stepped down the compression ratio to 2X (50% of the original connections remaining) the significant based pruning accuracy loss dropped to less than 1% from the original inference. Han et al. [4] used L1 normalization and L2 Normalization to separate the unimportant connections from important connections in Convolutional Neural Networks, and used magnitude based pruning to remove the unimportant connections and retrain pruned network to gain the original accuracy back. Our method on significant based pruning removes the important connections from the inference computational DAG and keep the accuracy loss minimal compared to magnitude based pruning without any retraining.

4.3 Significant based reordering accuracy

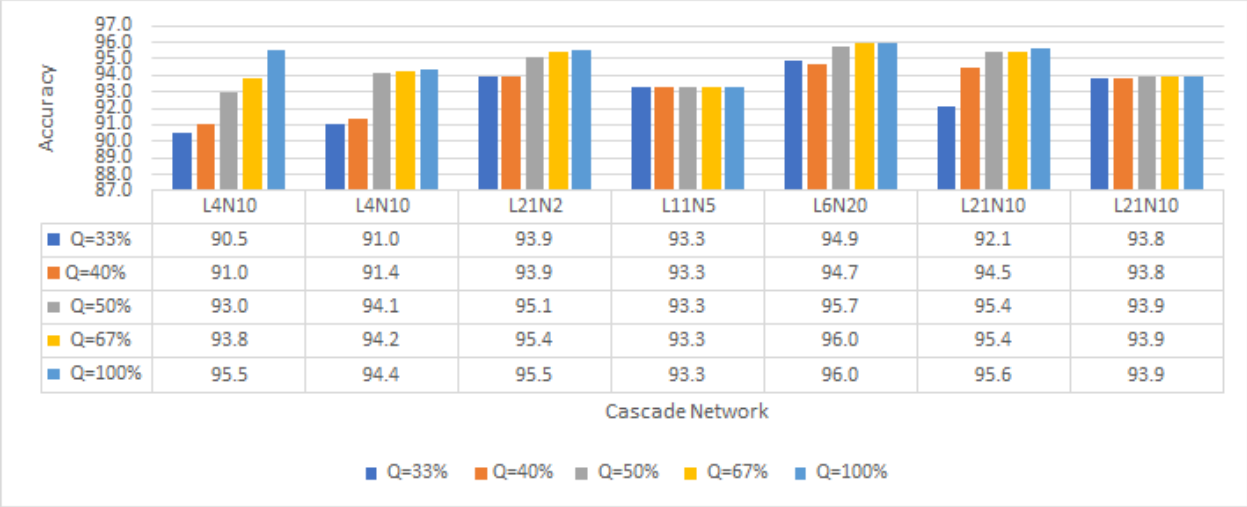


	L3N30	L3N40	L5N40	L6N40	L2N50	L3N50	L5N50	L4N15	L5N20	L6N20	L7N20	L5N25	L4N30	L5N30	L4N25	L6N30	L7N30
Q=33%	84.9	88.7	88.2	90.1	93.0	88.5	91.5	90.7	93.0	91.4	92.2	92.4	93.5	93.3	96.0	89.8	89.3
Q=40%	88.4	92.4	92.3	93.4	94.4	92.4	93.7	93.7	94.9	92.0	94.2	94.8	95.0	93.0	95.0	93.2	92.0
Q=50%	92.7	92.7	93.5	94.0	95.0	93.1	94.1	94.9	95.0	94.7	94.4	94.8	95.0	95.3	95.0	94.3	95.2
Q=67%	93.7	92.8	93.4	93.8	94.7	94.5	94.9	94.9	95.0	94.0	94.5	94.8	95.0	95.5	95.0	94.8	95.4
Q=100%	94.2	93.6	93.0	94.0	94.7	94.3	95.1	94.7	95.6	94.0	94.7	94.8	95.0	95.5	95.0	94.8	95.2

Figure 4.3: Prediction accuracy comparison between different pruning ratios

The Fig.4.3 shows the Fully connected classical Deep Neural Network's accuracy after significance based pruning, Q=33% is obtained by reordered the network into 3 sets of operations and executing the 1^{st} significant set and Q=40% is obtained by executing the first 4 sets from the program reordered into 10 sets. 100% represents the original inference program without pruning.

LxNy represents x layers classical fully connected deep neural network. The network L4N25 when pruned to 33% predicts with 96%, which is better than the original inference accuracy of 95%, which is achieved from less over-fitting which is a side effect of pruning.



	L4N10	L4N10	L21N2	L11N5	L6N20	L21N10	L21N10
Q=33%	90.5	91.0	93.9	93.3	94.9	92.1	93.8
Q=40%	91.0	91.4	93.9	93.3	94.7	94.5	93.8
Q=50%	93.0	94.1	95.1	93.3	95.7	95.4	93.9
Q=67%	93.8	94.2	95.4	93.3	96.0	95.4	93.9
Q=100%	95.5	94.4	95.5	93.3	96.0	95.6	93.9

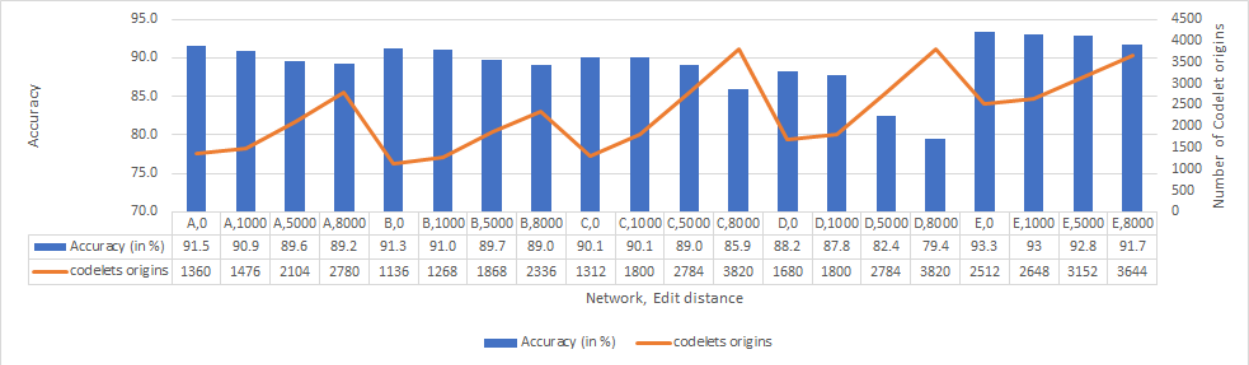
Figure 4.4: Prediction accuracy comparison for Cascade Network

The Fig.4.4 shows the prediction accuracy of the cascade neural networks. Where L4N10 describes a network with 4 hidden layers and each layer has 10 output neurons. In cascade neural network, each layer's output is fed into all the subsequent layers. Eg. 1^{st} layer is fed from the input image, the 2^{nd} layer in the cascade network is fed from the input image and the 1^{st} layer's output, and the 3^{rd} layer is fed with input image and 1^{st} and 2^{nd} layers output.

In most of the cascade networks, we can achieve very similar accuracy as the original inference program while executing less than 50% of the operations. Since the L4N10 has smaller number of parameters than other networks to classify the image, pruning $2/3$ of the computations results in larger accuracy drop than others. Other larger networks have more redundant computation and pruning $2/3$ of the computations is still sufficient to classify the image.

4.4 Codelets aware reordering accuracy

Fig. 4.5 shows the accuracy when executing the first 33% operations from codelet aware reordering while modifying the edit distance on 5 networks (Table 4.1). Edit distance 0 shows the original significance based reordering.



	A,0	A,1000	A,5000	A,8000	B,0	B,1000	B,5000	B,8000	C,0	C,1000	C,5000	C,8000	D,0	D,1000	D,5000	D,8000	E,0	E,1000	E,5000	E,8000
Accuracy (in %)	91.5	90.9	89.6	89.2	91.3	91.0	89.7	89.0	90.1	90.1	89.0	85.9	88.2	87.8	82.4	79.4	93.3	93	92.8	91.7
codelets origins	1360	1476	2104	2780	1136	1268	1868	2336	1312	1800	2784	3820	1680	1800	2784	3820	2512	2648	3152	3644

Figure 4.5: Accuracy of codelet aware reordering for Q=33%

By controlling the edit distance, the number of iterations that can be fit into codelets can be controlled. Which gives the flexibility to produces better implementation of piecewise regular program 3.3 with some accuracy loss. In network C, the total number of iterations that can be fit into codelets (Table 4.2) can be increased almost 3X by loosing 4.2% accuracy.

Table 4.2 lists the codelet shapes explored. The codelet shape contains 7 values, dimension of the codelet, followed by length of codelet in each dimension and lastly the stride along each dimension of the codelet. Consider Fig.4.6, codelet shape represents iterations in first dimension (l), second dimension (i) and the third dimension (j). The first shape in Table 4.2 represents 8 iterations along i in the inference program.

```
for l in 0..L
    for i in 1..I
        for j in 1..J
            S: y[l+1][i]+=w[l][i][j]*y[l][j]
```

Input program

Figure 4.6: Example Inference

Network	Number of Connections in the original network
A:L5N50	47410
B:L6N50	49960
C:L6N40	38370
D:L5N40	36730
E:L5N70	88570

Table 4.1: Network information

codelet shape	ed=0	ed=1000	ed=5000	ed=8000
3,1,8,1,1,1,1	2	2	19	76
3,1,8,1,1,2,1	2	4	27	43
3,1,1,8,1,1,1	23	22	39	40
3,1,1,8,1,1,2	1	3	6	11
3,1,4,4,1,1,1	1	1	0	0
3,1,4,4,1,2,2	0	0	0	0
3,1,4,1,1,1,1	87	108	178	216
3,1,4,1,1,2,1	59	61	107	124
3,1,1,4,1,1,1	108	107	108	87
3,1,1,4,1,1,2	14	14	21	24

Table 4.2: Codelet shapes found in Network C for different edit distances

Codelet aware reordering reorder operations based on the codelet shape provided, and the order of the shapes in the file impact the output reordered program.

Chapter 5

Conclusion

The scope of artificial neural networks have been growing drastically in the recent years. While the neural networks becoming more powerful, the network requires more memory and computation resources, The future applications demanding low power, higher throughput and low latency implementations on embedded resource constrained environment. The network pruning provides a solution to reduce the network size and efficient implementation on resource constrained environments.

We present a new methodology to reduce the inference network size by removing insignificant computations. We assign rank to each computation in the network based on the contribution significance to the final result and depending on the resource constraint or operation budget, we generate a size reduced inference program from the high ranking operations. We have showed in some classical feed forward networks, after removing more than half of the operations from the fully connected layers, the network still be able to classify with minimal accuracy loss. We also presented a method to control the standard structures sparse network, which enables to control the trade off between accuracy and implementation performance.

Future work includes, extending the significant based pruning to convolutional layers and explore the impact of channel level, filter level pruning, quantization and fixed point implementation.

Bibliography

[1] J. Leitner, S. Harding, M. Frank, A. Förster, and J. Schmidhuber. Artificial neural networks for spatial perception: Towards visual object localisation in humanoid robots. In *The 2013 International Joint Conference on Neural Networks (IJCNN)*, pages 1–7, Aug 2013.

[2] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, Nov 1998.

[3] Yann Le Cun, John S. Denker, and Sara A. Solla. Advances in neural information processing systems 2. chapter Optimal Brain Damage, pages 598–605. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1990.

[4] Song Han, Jeff Pool, John Tran, and William J. Dally. Learning both weights and connections for efficient neural networks. *CoRR*, abs/1506.02626, 2015.

[5] A. Polyak and L. Wolf. Channel-level acceleration of deep face representations. *IEEE Access*, 3:2163–2175, 2015.

[6] Wei Wen, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. Learning structured sparsity in deep neural networks. *CoRR*, abs/1608.03665, 2016.

[7] Zhuang Liu, Jianguo Li, Zhiqiang Shen, Gao Huang, Shoumeng Yan, and Changshui Zhang. Learning efficient convolutional networks through network slimming. *CoRR*, abs/1708.06519, 2017.

[8] Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. Pruning filters for efficient convnets. *CoRR*, abs/1608.08710, 2016.

[9] T. N. Sainath, B. Kingsbury, V. Sindhwani, E. Arisoy, and B. Ramabhadran. Low-rank matrix factorization for deep neural network training with high-dimensional output targets. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 6655–6659, May 2013.

[10] Emily L Denton, Wojciech Zaremba, Joan Bruna, Yann LeCun, and Rob Fergus. Exploiting linear structure within convolutional networks for efficient evaluation. In Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 27*, pages 1269–1277. Curran Associates, Inc., 2014.

[11] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. Deep learning with limited numerical precision. *CoRR*, abs/1502.02551, 2015.

[12] Naigang Wang, Jungwook Choi, Daniel Brand, Chia-Yu Chen, and Kailash Gopalakrishnan. Training deep neural networks with 8-bit floating point numbers. *CoRR*, abs/1812.08011, 2018.

[13] Song Han, Huizi Mao, and William J. Dally. Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding. *CoRR*, abs/1510.00149, 2015.

[14] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A. Horowitz, and William J. Dally. EIE: Efficient Inference Engine on Compressed Deep Neural Network. *arXiv e-prints*, page arXiv:1602.01528, Feb 2016.

[15] Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman. Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps. *arXiv e-prints*, page arXiv:1312.6034, Dec 2013.

[16] Travis Augustine, Janarthanan Sarma, Louis-Noël Pouchet, and Gabriel Rodríguez. Generating piecewise-regular code from irregular structures. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, pages 625–639, New York, NY, USA, 2019. ACM.

[17] The mnist database of handwritten digits. http://yann.lecun.com/exdb/mnist/. Accessed: 2019-11-25.

[18] Pytorch. http://https://pytorch.org. Accessed: 2019-11-25.