EXPLORATION OF ENERGY EFFICIENT HARDWARE AND ALGORITHMS FOR DEEP LEARNING

A Dissertation

Submitted to the Faculty

of

Purdue University

by

Syed Shakib Sarwar

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

May 2019

Purdue University

West Lafayette, Indiana

THE PURDUE UNIVERSITY GRADUATE SCHOOL
STATEMENT OF DISSERTATION APPROVAL

Dr. Kaushik Roy, Chair

Department of Electrical and Computer Engineering

Dr. Ananad Raghunathan

Department of Electrical and Computer Engineering

Dr. Vijay Raghunathan

Department of Electrical and Computer Engineering

Dr. Byunghoo Jung

Department of Electrical and Computer Engineering

Approved by:

Dr. Pedro Irazoqui

Head of the Departmental Graduate Program

Dedicated to my wife and my parents for their unconditional love and support.

ACKNOWLEDGMENTS

I want to thank my PhD supervisor, Prof. Kaushik Roy, who has become a role model for me. He guided me in every possible way to make me a better researcher. He was very patient and understanding while guiding me through tough periods with my research. I am grateful for his advices and efforts in shaping me into a researcher. His qualities as a supervisor as well as a person never cease to amaze me. I will always remember the fruitful conversations and the time spent with him that helped me throughout my PhD life.

I would also like to thank my PhD advisory committee members, Prof. Anand Raghunathan, Prof. Byunghoo Jung and Prof. Vijay Raghunathan for their insightful advice regarding my research. I also want to thank all the members of our research group, Nanoelectronics Research Laboratory, for helping in every possible way. I definitely had plenty of fun interacting with such a lively group of people.

I would like to take this opportunity to thank my family, and in particular, my parents and my wife for their unconditional love and support. My mother has always been the source of my inspiration. She always believed in me, even in the worst of times. My father is the origin of my perseverance and devotion towards my work. I always look up to him and take his advice. My wife has been a constant support for me. She joined with me in the middle of my doctoral studies and halved my struggle by sharing the load I faced in every aspect as a PhD student. Without her unconditional support, this work would never have been possible.

This thesis has been possible due to the help and support of numerous people and I want to apologize for not being able to list everyone here. I would like to express my gratitude to everyone who has helped me in pursuing my dream and try to be a better human being through my contribution to Science and Society.

TABLE OF CONTENTS

Page

Page

Page

LIST OF TABLES

Table Page

LIST OF FIGURES

Figure Page

ABSTRACT

Sarwar, Syed Shakib Ph.D., Purdue University, May 2019. Exploration of Energy Efficient Hardware and Algorithms for Deep Learning. Major Professor: Roy K. Professor.

Deep Neural Networks (DNNs) have emerged as the state-of-the-art technique in a wide range of machine learning tasks for analytics and computer vision in the next generation of embedded (mobile, IoT, wearable) devices. Despite their success, they suffer from high energy requirements both in inference and training. In recent years, the inherent error resiliency of DNNs has been exploited by introducing approximations at either the algorithmic or the hardware levels (individually) to obtain energy savings while incurring tolerable accuracy degradation. We perform a comprehensive analysis to determine the effectiveness of cross-layer approximations for the energy-efficient realization of large-scale DNNs. Our experiments on recognition benchmarks show that cross-layer approximation provides substantial improvements in energy efficiency for different accuracy/quality requirements. Furthermore, we propose a synergistic framework for combining the approximation techniques. To reduce the training complexity of Deep Convolutional Neural Networks (DCNN), we replace certain weight kernels of convolutional layers with Gabor filters. The convolutional layers use the Gabor filters as fixed weight kernels, which extracts intrinsic features, with regular trainable weight kernels. This combination creates a balanced system that gives better training performance in terms of energy and time, compared to the standalone Deep CNN (without any Gabor kernels), in exchange for tolerable accuracy degradation. We also explore an efficient training methodology and incrementally growing a DCNN to allow new classes to be learned while sharing part of the base network. Our approach is an end-to-end learning framework, where we focus

on reducing the incremental training complexity while achieving accuracy close to the upper-bound without using any of the old training samples. We have also explored spiking neural networks for energy-efficiency. Training of deep spiking neural networks from direct spike inputs is difficult since its temporal dynamics are not well suited for standard supervision based training algorithms used to train DNNs. We propose a spike-based backpropagation training methodology for state-of-the-art deep Spiking Neural Network (SNN) architectures. This methodology enables real-time training in deep SNNs while achieving comparable inference accuracies on standard image recognition tasks.

1. INTRODUCTION

The human brain is being studied for more than thousands of years. With the advent of modern technology, researchers were very much tempted to try to harness the amazing capabilities of the brain. In 1943, Warren McCulloch, a neurophysiologist, and a young mathematician, Walter Pitts, wrote a paper on how neurons might work [1]. This work is considered as the first step toward artificial neural networks. They modeled a simple Neural Network (NN) with electrical circuits using available electronic devices. As computers began to evolve from their infancy during the 1950s, researchers were able to model the rudiments of these theories concerning human brain. Nathaniel Rochester from the IBM research laboratories was the first person to simulate a neural network [2]. However, during the same time period, traditional computing began to flourish and, as it did, the emphasis in neural computing declined but not diminished. In 1957, John von Neumann suggested imitating simple neuron functions by using vacuum tubes or telegraph relays [3]. Also, Frank Rosenblatt, being intrigued with the operation of the eye of a fly, started working on the Perceptron [4]. Major portion of the processing, which tells a fly when to flee, is done in its eye. The Perceptron model resulted from this research was built in hardware. It is the oldest neural network still in use today. A single perceptron computes a weighted sum of the inputs, subtracts a threshold, and produces one of the two possible values as the result. A single-layer perceptron was found to be suitable in classifying a continuous-valued set of inputs into one of two classes. However, Minsky and Papert's analysis of perceptrons showed that a single layer perceptron cannot learn an XOR function, since classes in XOR are not linearly separable [5]. They also argued that it had to be done with multiple layers of perceptrons. Unfortunately, the perceptrons at that time had very limited learning capacity as the existing training algorithm could train only a single layer perceptron network [5].

In the early 1960s, 'Backpropagation' algorithm was developed by multiple researchers. However, Paul Werbos was first to propose that it could be used for training neural networks in his PhD Thesis in 1974 [6]. Then, in 1986, Rumelhart, Hinton and Williams [7] showed experimentally that this method can generate useful internal representations of incoming data in hidden layers of neural networks. However, further development of Neural Networks was heavily impeded till early 1990's due to another obstacle, lack of computing power.

The much needed boost for neural networks came in the year 1998, when the very first Convolutional Neural Networks (CNN) was demonstrated that lifted the field of Deep Learning. This pioneering work was named LeNet5 [8] after the lead researcher Yann LeCun. The LeNet5 architecture was elementary in developing the insight that image features are spatially distributed across the entire image. At that time, there was no Graphical Processing Unit (GPU) to help training, and even Central Processing Units (CPU) were slow. This work showed that convolutions are an effective way to extract similar features at multiple locations at the expense of few parameters. This was enormously useful considering the computation power limitations. LeNet5 is the origin of most of the recent architectures, and greatly inspired majority of the researchers in the field. However, from 1998 till 2010, neural network research was again in incubation due to lack of resources. During this period, most people did not notice the increasing power of neural networks, while many other researchers slowly progressed in the shadows. Data availability increased exponentially because of the rise of cell-phone cameras and cheap digital cameras. Computing power was also on the rise, CPUs were becoming faster, and GPUs became a general-purpose computing tool. These trends helped neural network research to progress, although at a slow rate. However, both increasing data and computing power allowed neural network to tackle the tasks more powerfully and accurately.

In 2012, Alex Krizhevsky proposed AlexNet [9], which won the difficult ImageNet [10] competition of classifying 1000 object classes by a large margin. AlexNet was a deeper and much wider version of the LeNet. In AlexNet, the author up-scaled

the insights of LeNet into a much larger neural network that could be used to learn much more complex objects and hierarchies. This also showed that GPUs can be efficiently utilized to supply the enormous computing power required for realizing deeper networks. From then on, a lot of new algorithms and network architectures have emerged to uplift the Deep Learning research.

1.1 Deep Learning and its Constraints

Deep Neural Networks (DNNs) are a class of brain-inspired machine learning algorithms that enable a computer to learn from observational data. *Deep Learning* is the field where these DNNs are applied and investigated. DNNs have demonstrated state-of-the-art results in a range of applications including image and speech recognition, natural language processing, and video analysis [11, 12]. Moreover, there have been several instances in the recent past where DNNs have been shown to outperform humans [13]. However, the memory and computational requirements of such large-scale networks are quite high. With energy consumption becoming a primary concern across all computing platforms, from data centers to mobile devices, energy-efficient realization of DNNs is of paramount interest. In fact, the vast energy overhead of large-scale DNNs has led to multi-dimensional research efforts spanning algorithms, architecture, circuits and devices, for the energy-efficient realization of DNNs [14–18]. At the algorithmic level, novel network topologies and models are being explored [15, 16], while at the hardware level, emerging devices capable of mimicking the neuronal and synaptic dynamics have been proposed [17, 18] to address the energy challenges.

Further, the quest for computing efficiency has led to the emergence of an alternate design paradigm, namely *approximate computing*, which exploits intrinsic algorithmic error resiliency for energy savings [19]. Significant benefits in energy can be obtained by judiciously approximating certain computations for negligible loss in output quality [20–24]. Neural networks are known to be highly error resilient [25, 26],

and are hence capable of tolerating approximations in the underlying computations. Researchers have explored the possibility of introducing approximations at different levels of DNN implementation in an effort to achieve energy efficiency for tolerable degradation in accuracy [27–32]. These research efforts on approximate computing for neural networks proposed and evaluated approximations at a single level of abstraction. For example, in [29,31,32], approximations were applied at the algorithmic level. Examples of approximations to DNNs at the hardware level include [28, 30]. These efforts have shown that some of the accuracy loss due to hardware approximations can be recovered by re-training the networks [27–29]. Since different approximations provide different energy-accuracy trade-offs, there is a need to compare these approximations and further explore whether better energy-quality trade-offs can be achieved through cross-layer approximation.

1.2 Contribution and Organization of the Thesis

The basic operation of Neural Networks consists of two phases, training and testing/inference. The training process is usually carried out off-line or in the cloud. The trained neural network is then used to process unseen data inputs. For large networks with millions of neurons, the testing process, although less compute-intensive than training, nevertheless requires significant computation. In this work, we explore and analyze approximations at different levels of abstraction and novel training algorithms for achieving energy efficiency during both in testing/inference and training, separately.

1.2.1 Improve Testing/Inference Energy-efficiency

To increase testing/inference energy-efficiency, we considered approximations at both Algorithm and Hardware level. At the algorithmic level, we explore low-complexity networks with fewer layers and/or neurons and perform pruning of synapses [32, 33]. At the hardware level, we utilize approximate multipliers [28] for neuronal compu-

tation, and approximate memory [30] for storing the synaptic weights. We systematically analyze the energy-accuracy trade-offs offered by different approximations in DNNs. The key contributions of our work are:

- We introduce different algorithmic and hardware level approximations in DNNs such as reduction in network complexity [for both Fully-Connected Networks (FCNs) and Convolutional Neural Networks (CNNs)], synaptic weight pruning, approximate multiplications [28, 34], and approximate read/write operations to memory [30]. We analyze the trade-offs between accuracy and energy consumption for standard datasets. We additionally show that retraining the approximate DNNs minimizes the accuracy degradation.

- We combine approximations at different levels to achieve improved energy benefits. We compare different approximations and their combinations in a quest for the best (lowest energy) configuration that meets a given accuracy specification [35].

- We develop a circuit to system-level simulation framework to evaluate the classification accuracy and the energy consumption of different DNN approximations.

1.2.2 Improve Training Energy-efficiency

For training, we considered approximations at Algorithm level only. To reduce the training complexity of Deep CNNs, we replace certain weight kernels of convolutional layers with Gabor filters [29]. The convolutional layers use the Gabor filters as fixed weight kernels, which extracts intrinsic features, with regular trainable weight kernels. This combination creates a balanced system that gives better training performance in terms of energy and time, compared to the standalone Deep CNN (without any Gabor kernels), in exchange for tolerable accuracy degradation. We also explore an efficient training methodology and incrementally growing a DCNN to allow new classes to be learned while sharing part of the base network. Our approach is an end-to-end

learning framework, where we focus on reducing the incremental training complexity while achieving comparable accuracy without using any of the old training samples. Finally, we propose a spike-based backpropagation training methodology for state-of-the-art deep Spiking Neural Network (SNN) architectures. This methodology enables real-time training in deep SNNs while achieving comparable inference accuracies on standard image recognition tasks.

The key contributions of our work are:

- We proposed the use of Gabor filters in different layers of the CNN to lower the computational complexity of training CNNs [29]. The novelty of our work lies in the fact that we introduced Gabor filters with regular trainable weight kernels in the intermediate layers of the CNN. We designed several Gabor filter based CNN configurations in order to get the best trade-off between accuracy and other parameters of interest, especially energy consumption. We show that the accuracy degradation can be mitigated by partially training the Gabor kernels, for a small fraction of the total training cycles.

- We explore an efficient training methodology and incrementally growing a DCNN to allow new tasks to be learned while sharing part of the base network [36]. We propose sharing of convolutional layers to reduce computational complexity while training a network to accommodate a new set of classes without forgetting the old tasks. We developed a methodology to identify optimal sharing of convolutional layers in order to get the best trade-off between accuracy and other parameters of interest, especially energy consumption, training time and memory access.

- We develop a spike-based supervised gradient descent BP algorithm that exploits a conditionally differentiable approximated activation function of LIF neuron [37]. In addition, we leverage the key idea of the successful deep ANN models such as LeNet5 [8], VGG [38] and ResNet [39] for efficiently constructing state-of-the-art deep SNN network architectures. We also adapt dropout [40]

technique in order to better regularize deep SNN training. We demonstrate the effectiveness of our methodology for visual recognition tasks on standard character and object datasets and a neuromorphic dataset.

We show that our proposed methodologies leads to energy efficiency, reduction in storage requirements and training time, with minimal degradation of classification accuracy.

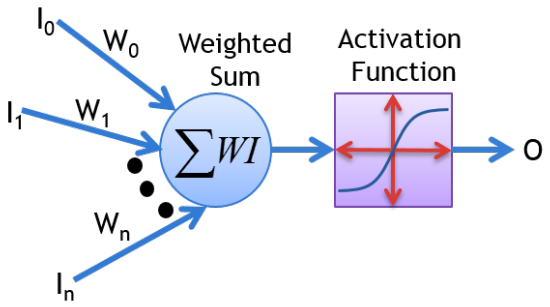
This dissertation is organized as follows. Chapter 2 describes the basics of neural networks used in this work. Chapter 3 and 4 explains the approximations at hardware and algorithm levels, respectively, for DNN inference. Chapter 5 discusses the combination of different approximations used to improve energy-efficiency during inference. Chapter 6 describes the application of Gabor filters as fixed convolutional kernels in deep CNNs. Chapter 7 explains an efficient training methodology of incrementally growing a DCNN and its benefits. Chapter 8 describes spike-based supervised gradient descent BP algorithm that can be utilized to train deep SNNs with supervision from direct spiking input. Finally, chapter 9 concludes the dissertation.

2. BASICS OF NEURAL COMPUTING

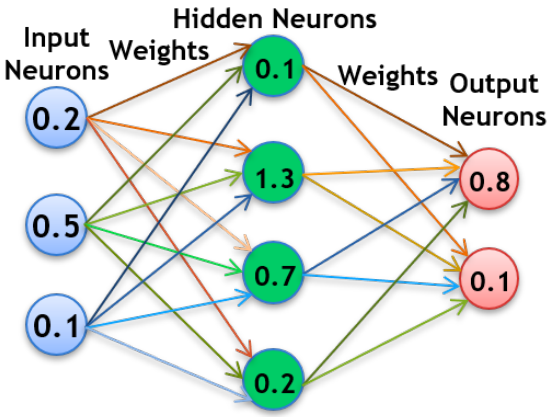
Neural Networks (NN) are a medium for neural computing. The basic operation of NNs consists of two phases, training and testing/inference. NNs can be divided into two major branches based on their operating principles, namely, Spiking Neural Network (SNN) and Non-spiking or Artificial Neural Network (ANN). Both ANN and SNN can be further divided in to sub-categories depending on their network connectivity. In this work, we consider two basic types of feed forward networks, fully-connected (FC) and convolutional neural networks (CNN) for studying multi-level approximations in NNs. In this chapter, we will go through these fundamental concepts.

2.1 ANN

The fundamental elements of an artificial neural network are neurons and synapses. The output of an artificial neuron is a weighted sum of its inputs passed through an activation function (Fig. 2.1). The activation function can be hard-limiting (*e.g.* *step* function) or soft-limiting (*e.g.* logistic *sigmoid* function, *tanh* function). Soft-limiting functions are preferred as they allow much more information to be communicated across neurons and greatly improve the neural network modeling capability while reducing network complexity. A network consisting of artificial neurons has real numbers as inputs and outputs of each neuron.



(a)



(b)

Fig. 2.1.: (a) Output of an artificial neuron is the weighted summation of its inputs passed through an activation function. (b) A network consisting of artificial neurons has real numbers as inputs and outputs of each neuron.

2.2 SNN

Spiking Neural Network (SNN), a third generation product of neural network models, provides greater level of realism in a neural simulation. The inputs to SNNs are spike trains (figure 2.2). One major feature of SNNs is the incorporation of temporal data into their operating model in addition to neuronal and synaptic states. The neurons in a SNN do not fire at each propagation cycle (as it happens with typical ANNs), rather fire only when a membrane potential (an intrinsic quality of the neuron

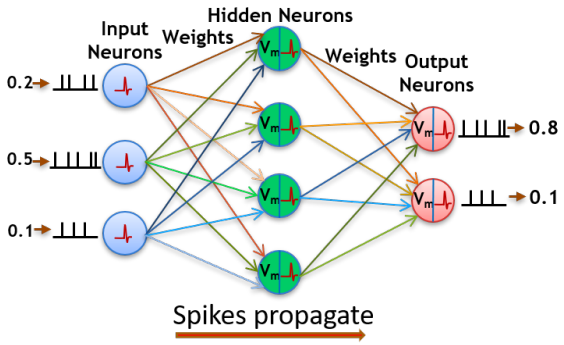


Fig. 2.2.: The inputs to SNNs are spike trains that propagate through the hidden layers all the way to the output neurons.

related to its membrane electrical charge) crosses a threshold. When a neuron fires, it generates a spike signal which travels to other neurons. The membrane potential of a spiking neuron is normally considered to be the neuron's state, with incoming spikes increasing its value, and then either firing or decaying over time. Inference in SNNs is based on accumulating sparse spiking events over time.

2.2.1 Fully-Connected Networks

In FCNs, the neurons are connected in an acyclic (feed-forward) manner as illustrated in Fig. 2.3. In such a NN, every neuron in a layer is connected to all the neurons in the following layer via synapses with unique individual connection weights.

2.2.2 Convolutional Neural Networks

CNNs consist of a hierarchical arrangement of alternating convolutional and spatial pooling layers followed by a fully-connected layer, with nonlinearity applied at the end of each layer. A typical architecture of a deep CNN is shown in Fig. 2.4. Convolutional layers extract complex high-level features, while spatial-pooling layers are used for dimensionality reduction and fully-connected layers are used for inference. To improve generalization and to reduce the number of trainable parameters,

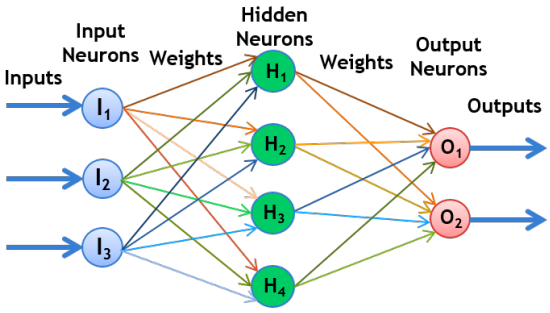


Fig. 2.3.: Feedforward FCN, where each neuron in a layer is connected to all the neurons in the following layer as shown by arrows. The different colored arrows indicate that each input is multiplied by different weights.

a convolution operation is exercised on small regions of input. One salient benefit of CNNs is the use of shared weights in convolutional layers, implying that the same filter (weight bank) is used for each pixel of the image; this reduces memory footprint and enhances performance.

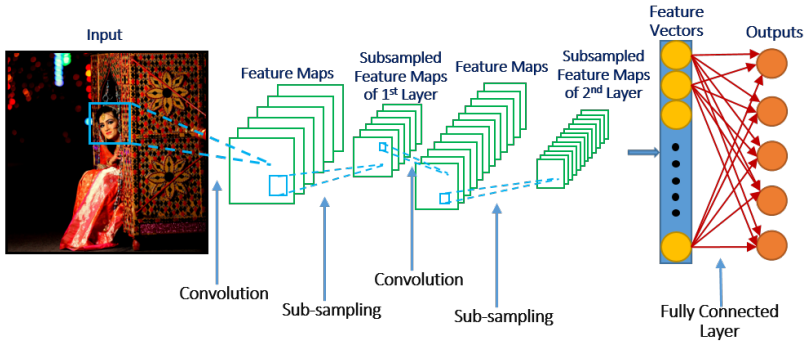


Fig. 2.4.: Architecture of a deep CNN.

2.3 Training

Neural networks have separate layers, connections, and directions of data propagation. In order to train a neural network, training data is put into the first layer of the network, and individual neurons are assigned a weighting to the input based on

how correctly the task is being performed or how much correlated the input is to the output. There are two popular methods for training a NN:

2.3.1 Back-propagation

Back-propagation, short for *'backward propagation of errors'*, is an algorithm for supervised learning using gradient descent [7]. Usually it is used for training artificial neural networks. Given an ANN and a specific error function, this method calculates the gradient of the error function with respect to the synaptic weights. The *'backwards'* part of the name originated from the fact that the calculation of the gradients proceeds backwards through the network. *i.e.* The gradients of the final layer of weights are calculated first and the gradients of the first layer of weights are calculated last. Part of the computations of the gradients of one layer is reused in the computation of the gradients for the immediate previous layer. This backwards flow of the error information provides an efficient way of computing the gradients at each layer compared to the naive approach of calculating the gradients of each layer separately.

Back-propagation method's popularity has experienced a resurgence due to the widespread use of deep neural networks. It is considered as one of the most efficient algorithms for Deep Learning, and modern implementations take advantage of specialized GPUs to further improve its performance.

2.3.2 STDP

Spike-timing-dependent plasticity (STDP) is a biological process closely related to the activity in brains. It refers to the adjustments of the strength of connections between neurons in the brain. This method is mostly used for training SNNs. Based on the relative timing of a particular neuron's output and input spikes (or action potentials), STDP process adjusts the connection strengths. This process partially

explains the activity-driven enhancement of nervous systems, especially the long-term potentiation and long-term depression processes.

In the STDP process, if an input spike to a neuron tends (on average) to occur immediately before that neuron's output spike, then the connections to that particular input is strengthened. If an input spike tends (on average) to occur immediately after an output spike, then the connections to that particular input is weakened. Hence it is called 'spike-timing-dependent plasticity' [41]. Thus, inputs that might be exciting the post-synaptic neuron are made even more likely to contribute in the future, whereas inputs that are generated after the post-synaptic spike are made less likely to contribute in the future. The process continues until only a fraction of the initial set of connections remain, while the influence of all others is diminished. When many of its inputs occur within a brief period, a neuron produces an output spike. Therefore, the fraction of connections with inputs that remain are those that tended to be correlated in time. Furthermore, since the connections of the inputs that occur before the output are strengthened, the inputs that provide the earliest indication of correlation will eventually become the most significant input to the neuron.

2.4 Inference

Inference is the process of taking real-world data and quickly producing a prediction about the class or correctness of the input. In neural networks, new unseen data input is propagated forward through an already trained network to test the performance (accuracy/quality) of the network.

In this work, we employ approximations at different levels of deep NNs (both Fully-connected and Convolutional Neural Networks in ANN domain) for achieving energy efficiency during inference/testing. These approximations are explained chapters 3-5.

3. APPROXIMATE HARDWARE DESIGN FOR DNNS

Neural networks are good candidates for approximate computing due to their inherent error resiliency. Therefore, several approximation techniques have been proposed for NNs. In this chapter, our focus is to explore approximations at the hardware levels.

3.1 Approximate Multiplier for DNNs

In this subsection, we discuss the approximations applied to the multiplication operation in the neurons of a deep neural network.

3.1.1 Motivation: Computation Energy Consumption by Multiplier

A neuron is a fundamental computational unit of an NN. Typically, a neuron performs a MAC operation (*i.e.*, integrates the product of the incoming inputs and synaptic weights) to obtain a weighted sum, followed by a non-linear activation on the weighted sum, to produce the resultant output. The most power consuming operation among the neuronal computations is multiplication, which by far outweighs the summation and activation operations. To address this issue, we used an approximate Alphabet Set Multiplier (ASM) proposed in [20] that achieves significant reduction in neuronal computation energy. In [28], computation sharing is used in conjunction with ASM to design energy-efficient hardware for inference.

3.1.2 Alphabet Set Multiplier

A multiplication operation can be decomposed into simple shift and add operations. The decomposition is based on the multiplicand *'W'*, which in our case

represented by the synaptic weights. Sample decomposition of two multiplication operations $W_1 \times I$ and $W_2 \times I$, are shown in Table 3.1.

Table 3.1.: Decomposition of Multiplication Operation

Weights	Decomposition of Product
$W_1 = 01101011_2(107_{10})$	$W_1 \times I = 2^5.(0011).I + 2^0.(1011).I$
$W_2 = 01001010_2(74_{10})$	$W_2 \times I = 2^6.(0001).I + 2^1.(0101).I$

Note that, in the table, few small bit sequences (0001_2, 0101_2, 0011_2, 1011_2) are multiplied with the input *'I'*. These small bit sequences, a_k are referred to as *alphabets*. If *I, 3I, 5I, 7I, 9I, 11I, 13I,* and *15I* are available, the entire multiplication is down to a few shift and add operations. Based on this insight, instead of multiplying the multiplier with the multiplicand, some lower order multiples of the input are shifted and added in ASM [20]. An ASM consists of a *pre-computer bank*, an *'adder'*, and one or more *'select'*, *'shift'* and *'control logic'* units. The *pre-computer bank* computes the lower order multiples of the input which are the products of the input and some pre-specified *alphabets*. These *alphabets* are collectively termed the *alphabet set* (denoted by {1,3,5,...}). Overall, the ASM has four steps: i) generate the products of the input and the *alphabets* ii) select a product iii) shift that product iv) add the shifted products. In this work, synaptic weights are taken as 4 bit, 8 bit and 12 bit words for neurons of equivalent bit precisions. For this section, we will consider only 8 bit synapses for explaining the operation of ASMs. The 8 bit word is divided into two quartets for the ASMs. This requires a final addition after select and shift operations. Based on the multiplicand, different combinations of select, shift and addition will occur, which will be controlled by a *'control logic'* unit. For performing general multiplication operation, all possible combinations must be covered. It has been shown that to cover these combinations, 8 alphabets {1,3,5,7,9,11,13,15} are required for bit sequence size of 4 bits [42]. It must be noted that the number of alphabets being used in the pre-computer bank directly translates to power dissipation, while

the number of communication buses (out of the pre-computer) is also proportional to the number of alphabets. However, in order to achieve lower routing complexity and power consumption, use of reduced number (less than 8) of alphabets is proposed in [28]. The reduction of number of alphabets is only possible owing to the error resilience of neural computing. Using Fig. 3.1, the working principle of an 8 bit 4 alphabet ASM is explained next.

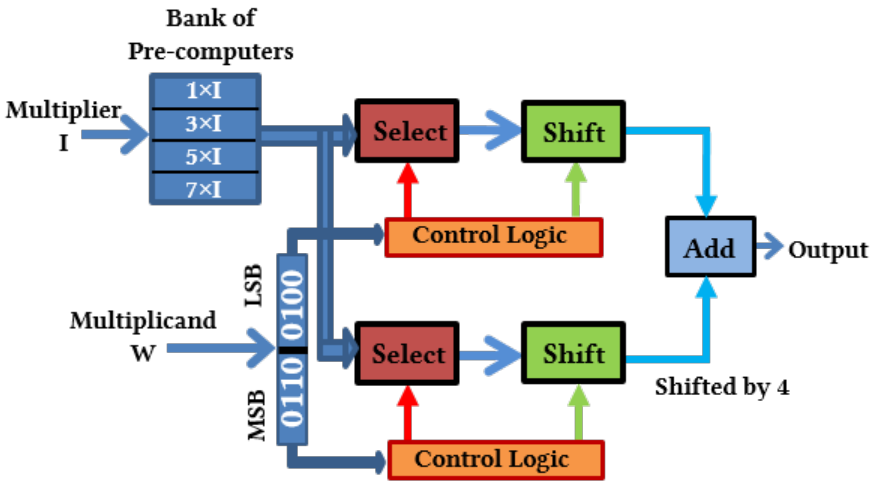


Fig. 3.1.: 8 bit 4 alphabet ASM (modified from [28]).

Multiplier *'I'* is supplied to the pre-computer bank which generates products of input and the alphabets. These products are realized by shift and add operations. In this example, the alphabet set is {1,3,5,7}. Hence, the pre-computer bank will generate *1I, 3I, 5I, 7I*. Multiplicand *'W'* is divided into two parts which will work as inputs for the *'control logic'* circuits. Based on the *'W'*, control signals for the *'select'* and *'shift'* units are generated by the respective *'control logic'* circuits. The *'select'* units select suitable products from the pre-computer bank and pass them to the *'shift'* units. *'Shift'* units shift the input by the required amount. Finally, the *'adder'* unit adds the two separate values to get the multiplication result. For instance, to realize the multiplication of Y=100 ($0110\ 0100_2$) and X, we have to generate 0100_2 X (4X) and 0110_2 X (6X) $\times\ 2^4$ (shifted by 4 corresponding to the relative bit position),

and sum them up. Note that 4X and 6X can be generated by selecting 1X and 3X from the pre-computer bank, and shifting them respectively by 2 bits and 1 bit. The multiplication decomposition is demonstrated by the following equation:

$$01100100_2 \times X = (3X \times 2^1) \times 2^4 + (1X \times 2^2) \times 2^0$$

3.1.3 Computation Sharing Multiplication

Since these ASMs require pre-computing unit and control circuitry, they will only be advantageous if they can be used in a distributed way with minimum number of alphabets, *i.e.* share the product of input and alphabets with several multiplication units. The CSHM [43] architecture can be used to serve that purpose. Fig. 3.2 illustrates a CSHM consisting of a common pre-computer bank, shared between a number of ASM based multiplication units.

In a feedforward ANN, each input is multiplied by a number of different weights to feed the different neurons (Figure 2.3). Therefore, the pre-computer bank can be shared between the neurons that are processing the product of same input but different weights in parallel. In this work, we used the processing unit implemented in [28], which processed four neurons at a time, thus making it possible for four ASM units to share the product of input and alphabets from a common pre-computer bank, as illustrated in Fig. 3.2.

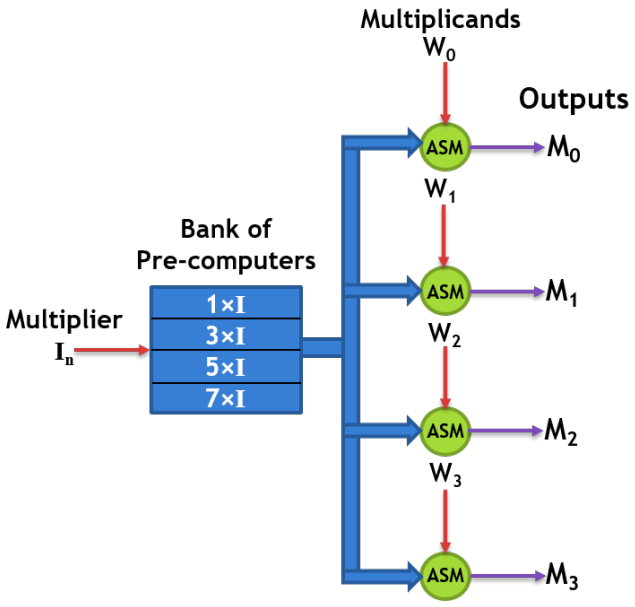


Fig. 3.2.: 4 alphabet ASMs using CSHM architecture (modified from [28]).

3.1.4 Selection of Good Alphabets

Use of reduced number (less than 8) of *alphabets* is proposed in [28] to reduce power dissipation of the *pre-computer unit*. However, reduction of the number of *alphabets* decreases flexibility of synapses during training, which in turns reduces the network accuracy. Therefore, proper selection of *alphabets* is very important to get maximum energy benefits with minimal loss of accuracy. While selecting *alphabets* for an *alphabet set*, it must be ensured that the *alphabet* can support higher number of bit shift operations to generate other bit quartets which are not in the *alphabet set*. For example, bit shift operation can be applied on *alphabet* 1 (0001_2) to generate 3 other bit quartets 2 (0010_2), 4 (0100_2) and 8 (1000_2). On the other hand, *alphabet* 3 (0011_2) can produce 2 other bit quartets, 6 (0110_2) and 12 (1100_2), while 5 (0101_2) and 7 (0111_2) each can produce only one other bit quartet 10 (1010_2) and 14 (1110_2), respectively. It is difficult to use *alphabets* $9(1100_2)$, $11(1011_2)$, $13(1101_2)$ and $15(1111_2)$, since they cannot generate any other bit quartets. Therefore, selection priority of *alphabets* is: $1 > 3 > 5, 7 > 9, 11, 13, 15$.

3.1.5 Design Approach & Methodology

The use of ASM to exploit error resilience, and sharing of *alphabets* are the bases of proposed approximate neurons. This section outlines the key ideas behind the proposed design methodology.

Application of Weight Constraints

To perform multiplication using ASM, *'select', 'shift'* and *'add'* operations in a number of different combinations needs to be performed. The efficacy of ASM mostly depends on the number of *alphabets* used to encompass the range of the combinations. If the bit sequence size of 4 bits is used for the decomposition of the multiplication operation, then an *alphabet set* of 8 alphabets {1,3,5,7,9,11,13,15} is sufficient to produce any product using the select, shift and add operations.

To achieve higher energy savings, the number of *alphabets* used in the proposed ASM is fewer than the quantity required for ideal (accurate) operation. As a result, it may not support all the multiplication combinations, which leads to approximations in multiplication. For example, when a 4 alphabet {1,3,5,7} ASM is used, we can generate 12 (including 0 (0000_2)) out of 16 possible combinations of 4 bits by bit shift operations (e.g. from 1 (0001_2) we get 2 (0010_2), 4 (0100_2) and 8 (1000_2)). It cannot produce the products, *9I, 11I, 13I* and *15I*, with (9,11,13,15) being the quartets from the synaptic weights. Therefore, we cannot generate the product $01101001_2 \times I$ with any select, shift and add combinations, since the LSB $1001_2(9_{10})$ is not supported by the *alphabet set*. However, to guarantee proper functioning of the neural network, it must be ensured that the unsupported multiplication combinations do not lead to significant computational errors. To address this issue, we introduce constrained training of the ANN so that these unsupported combinations never occur. Since ANN applications are inherently error resilient, we can exploit this and get favorable set of weights. For this purpose, the synaptic weights (9,11,13,15) are restricted to the nearest supported values (8,10,12,14). This is similar in effect to quantization, which

drops some amount of information resulting in accuracy degradation. To salvage some of the lost accuracy, the network needs to be retrained with the imposed constraints. The retraining overhead is marginal compared to the original training.

In a NN, the synaptic weights can be either positive or negative, while regular primary input, for example, image pixel value is usually non-negative. Also, if we use non-negative activation function ('sigmoid' or 'ReLU' [44]), then the inter-layer inputs will be non-negative too. For multiplication, we used the magnitude of the synaptic weights. For storing the sign information of synaptic weights, one extra bit is used. The sign bit also determines the sign of the multiplication output. If negative input is also present, then one extra bit will be required to store that information and the sign of the multiplication output will depend on both the input sign and synaptic weight sign bits.

Next, as an example, the algorithm for constraining weights for 12 bit ASM is explained. Here, we consider a 12 bit unsigned synaptic weight as a concatenated version of 3 bit quartets P, Q and R, where P is the MSB and R is the LSB (shown in Fig. 3.3). P, Q and R each can have 16 combinations, 0 (0000_2) to 15 (1111_2). If we use 2 alphabets {1,3} only, the maximum number of supported combinations out of the 16 is 8. In that case, we cannot support 5_{16}, 7_{16}, 9_{16}, A_{16}, B_{16}, D_{16}, E_{16}, F_{16} for P, Q and R. Hence, we convert those unsupported values to the nearest supported value ensuring minimum loss in precision. Algorithm 1 is the weight constraint mechanism for 12 bit 2 Alphabets {1,3} Multiplier.

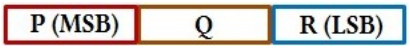


Fig. 3.3.: 12 bit weight value decomposed into three quartets (modified from [28]).

ALGORITHM 1: Weight constraint for 12 bit 2 Alphabets {1,3} Multiplier

Input: Absolute weight value PQR, list of unsupported quartets values $\{unsV\}$, sign information of the weight value *'s'*.

Output: Updated Weight value PQR_{new}

1. If P=$\{unsV\}$
2. If Q=$\{unsV\}$
3. If R=$\{unsV\}$, then round-down R,
4. based on R_{new} round-up/down QR,
5. based on Q_{new} round-up/down PQR
6. Else based on R round-up/down QR,
7. based on Q_{new} round-up/down PQR
8. Else based on Q round-up/down PQR
9. Else if Q=$\{unsV\}$
10. If R=$\{unsV\}$, then round-down R,
11. based on R_{new} round-up/down QR,
12. based on Q_{new} round-up/down PQR
13. Else based on R round-up/down QR,
14. based on Q_{new} round-up/down PQR
15. Else if R=$\{unsV\}$, then round-down R,
16. based on R_{new} round-up/down QR,
17. based on Q_{new} round-up/down PQR
18. If s=0, Return PQR_{new}
19. Else Return *2*'s complement of PQR_{new}

Rounding Logic

For approximate multiplication operation, we must round-up/down an unsupported value to the nearest supported value ensuring minimum loss of information. For every two consecutive supported values, the average of them is considered as the threshold point for rounding. For a 4 bit synapse, consider the two consecutive supported values 8_{16} and C_{16} (using only the alphabets {1,3}); then the threshold is

(8_{16}+ C_{16})/2=A_{16}. If the unsupported value 9_{16} comes up, we will convert it to 8_{16}, else if A_{16} or B_{16} comes up, we will convert it to C_{16} (Fig. 3.4a).

(a) (b)

Fig. 3.4.: For 2 alphabets {1,3} ASM, rounding up/down the unsupported values (a) 4 bit synapse, (b) 8 bit synapse.

The threshold point for rounding is different for different unsupported values. If the unsupported value is between two supported values, then based on the threshold point it can be converted to appropriate supported value. But there are cases which do not fall in this pattern. For example, if the unsupported value is E_{16}, then it has only one nearest supported value C_{16}. If we use 4 bit synapse, then there is no option other than converting E_{16} to C_{16}. But if we use 8 bit or 12 bit synapse, then we can take help from the upper 4 bits in this situation. This is shown in Fig. 3.4b. Here, we have to consider the upper 4 bits as well for finding the nearest supported values and threshold point.

Neural Network Design Methodology

With help of Fig. 3.5, algorithm 2 describes the overall NN training and testing methodology. The inputs are a neural network (NN), its corresponding training dataset (TrData), testing dataset (TsData), and a quality constraint (Q) that determines the minimum acceptable quality in the implementation. The quality specifications are application-specific.

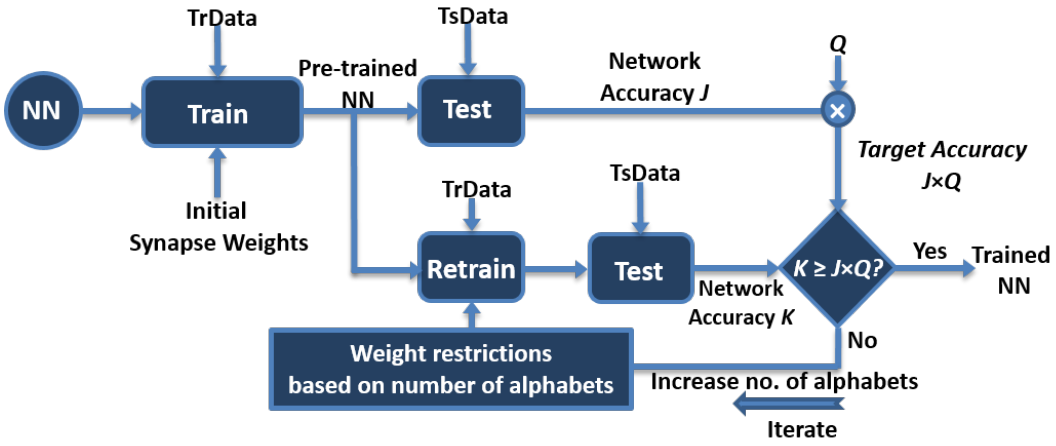


Fig. 3.5.: Overview of the ANN design methodology (modified from [28]).

ALGORITHM 2: NN training and testing methodology

Input: Neural network: NN, Training dataset: TrData, Testing dataset: TsData, Quality constraint: Q≤1.

Output: Retrained NN meeting the quality constraint.

1. Train the NN using TrData without any weight constraints till the training reaches near saturation, *i.e.* minuscule improvement in recognition accuracy can be achieved through more training.
2. Test the network using the TsData to get the network accuracy J. Create a restore point.
3. Retrain the network imposing constraints for minimum number of alphabets (start with 1) on weight update with lower learning rate till it again reaches near saturation.
4. Test the retrained network to find the new network accuracy K and compare the network accuracy using J, K and Q.

   If accuracy is satisfactory, *i.e.* if $K \geq J \times Q$, then end the training.

   Else restart from the restore point created in 2 and repeat steps 3 and 4 with increased number of alphabets.

Retraining

Retraining could effectively be used to mitigate the accuracy degradation incurred due to approximations in multiplication. Adapting the learning rate is vital for efficiently retraining the network with these approximations in place. The learning rate is basically a multiplication factor in the weight update equation [45] that influences the speed and quality of learning. Precise adjustment of the learning rate is needed, when using the approximate multiplier, due to the non-uniformity in the distance between the allowed weight levels. The non-uniformity is illustrated by the following example. Let us consider a 2 alphabet {1,3} ASM, where the acceptable weight levels are 0x, 1x, 2x, 3x, 4x, 6x, 8x, and 12x. It can be deduced that the distance between the levels 8x and 12x is 4x, while that between 6x and 8x is 2x. In this scenario, if the learning rate is too low, the updates might not be substantial enough for the weights to overcome the distance barrier between certain allowed levels. This arises the possibility of the weights getting stuck at a specific level (that may potentially lead to non-convergence in training), which is unfavorable to the learning process. On the contrary, too high a learning rate might cause the weights to widely oscillate between the different levels, which leads to accuracy deterioration. Hence, the determination of the optimal learning rate for retraining the approximate NN is of extreme importance.

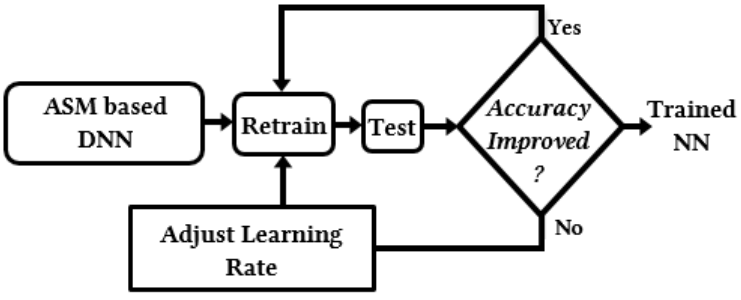


Fig. 3.6.: Flow diagram of the retraining process for an ASM based DNN.

Fig. 3.6 illustrates the flow diagram of the retraining process. The retraining begins with the highest learning rate that was used to start training the NN without approximation. With the weight restrictions in place, if the accuracy improves, retraining is carried on with the same learning rate for a few more iterations, until there is no noticeable improvement in the accuracy. On the other hand, if the accuracy does not improve, the learning rate is adjusted and the NN is further retrained. The adjustment of learning rate is carried out by reducing it by some factor. The process of adjusting the learning rate is continued until the accuracy improvement saturates. Point to be noted here is that the overhead for retraining an approximate NN for few epochs is negligible compared to the number of epochs required to train the original NN (without approximations).

3.1.6 Multiplier-less Neuron (MAN)

From the accuracy results of ASM in artificial neurons, we observed that even with only 1 alphabet $\{1\}$ in all layers, we are able to achieve accuracy within $\sim$0.5% of conventional implementation. The additional advantage of using only 1 alphabet, specifically $\{1\}$, is that we do not have to generate any alphabet set, the input only is sufficient for the 1 $\{1\}$ alphabet requirement. That means we do not need multiplication, only shifting and adding is enough. This eliminates the necessity of the *pre-computer bank* and *alphabet 'select'* unit, leading to a 'Multiplier-less' neuron (Fig. 3.7).

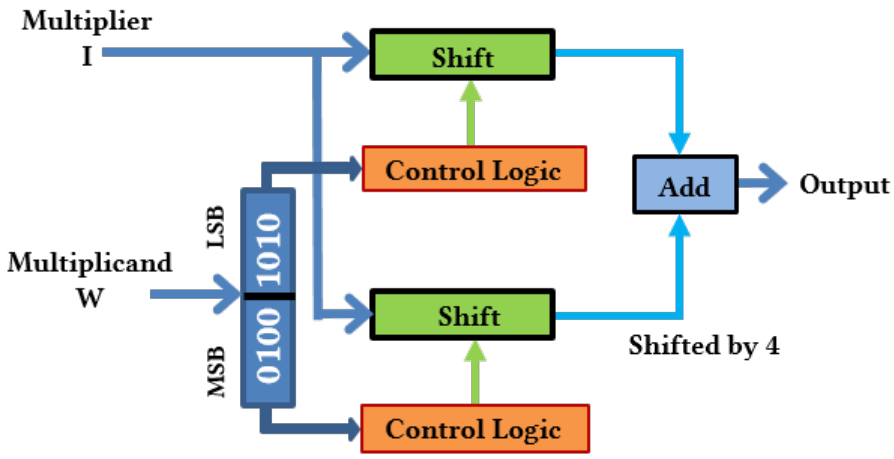


Fig. 3.7.: 8 bit 1 alphabet {1} ASM (MAN) (modified from [28]).

3.1.7 Realization

In this section, we discuss the circuit to system-level simulation framework that we developed to analyze the effectiveness of the approximations on NNs. For the approximate multiplication, we implemented the multiplier and adder units at the Register-Transfer Level (RTL) in Verilog, and mapped the designs to the 45nm technology library using the Synopsys Design Compiler. We subsequently built a neuronal energy computation model based on the number of MAC (multiply-accumulate) operations in the forward propagation of the NN algorithms (FCN and CNN). The power and delay numbers of the individual adder and multiplier units obtained from the Design Compiler are subsequently fed to the energy computation model to estimate the total neuronal energy consumption.

At the system-level, the deep learning toolbox [45] and MatConvNet [46], which are MATLAB based open source neural network simulators, are used to model the approximations and evaluate the performance (classification accuracy) of the DNNs under consideration. We implemented the fully connected and convolutional NNs without data augmentation, batch normalization, and dropout features to primarily

single out the effects of different approximations. Details of the benchmarks used in our experiments are listed in Table 3.2.

Table 3.2.: Benchmarks

Application	Dataset	NN Model	# Training Samples	# Testing Samples
Digit Recog.	MNIST	FCN	60000	10000
Object Recog.	CIFAR 10	CNN	50000	10000

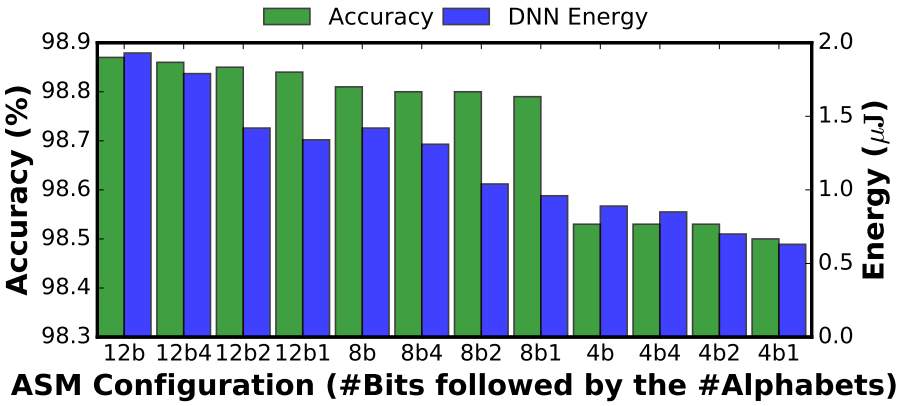
3.1.8 Results

We now present the computational energy-accuracy trade-offs offered by the ASM based deep FCNs and CNNs, and perform a comparison with the conventional (unapproximated) DNNs.

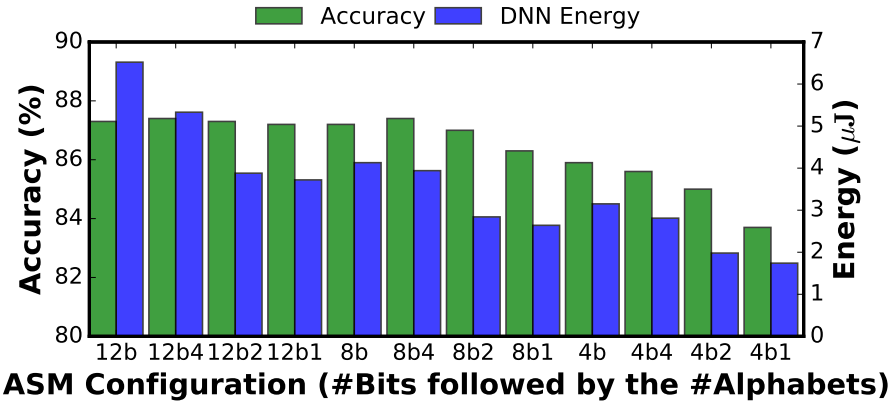
We observe from Fig. 3.8a that, ASM based FCNs (trained on MNIST) using 4 alphabets {1,3,5,7} do not provide much energy savings over conventional (unapproximated) DNNs of 12, 8 and 4 bit neurons. On the other hand, 18-27% reduction in energy consumption is achieved using only 2 {1,3} alphabets ASM based FCNs. In the case of multiplier-less neurons (using only 1 alphabet {1}), 26-32% reduction in energy consumption is achieved. The accuracy degradation is less than 0.40% for ASM based DNNs compared to 12 bit conventional DNN baseline.

Fig. 3.8b shows that, 5-18% and 31-40% energy savings is achieved using 4 alphabet and 2 alphabet ASM based CNNs (trained on CIFAR10) respectively. Our analysis further indicates that up to 44% energy savings is achieved using multiplier-less neurons (using only 1 alphabet {1}). We note that although the 4-bit ASM based CNNs offer the lowest energy consumption relative to other ASM configurations, they lead to higher degradation in accuracy (up to 2.3%), which is undesirable. The higher accuracy loss can be attributed mostly to lower bit precision. From the effect of bit

precision scaling, we found that a Deep CNN [1024×3 $(5 \times 5)192c$ $160fc$ $96fc$ $(3 \times 3)mp$ $(5 \times 5)192c$ $192fc$ $192fc$ $(3 \times 3)mp$ $(3 \times 3)192c$ $192fc$ $10o$] of 4 bit precision trained on CIFAR10 can achieve only an accuracy of 86% compared to the baseline (12 bit precision) of 87.3%. If hardware approximations are applied on this network, further accuracy degradation is unavoidable.



(a)



(b)

Fig. 3.8.: Energy/accuracy trade-off comparison between ASM based DNNs and Conventional DNNs, for (a) MNIST dataset on FCN and (b) CIFAR 10 dataset on CNN. 12b: 12 bit synapse NN, 12b4: 12 bit synapse NN with 4 alphabet ASM, 12b2: 12 bit synapse NN with 2 alphabet ASM, 12b1: 12 bit synapse NN with 1 alphabet ASM. Similar notations for 8 bit and 4 bit NNs.

3.2 Approximate Memory for DNNs

In this section, we discuss the approximations applied to weighted synapses interconnecting various layers of a deep neural network.

3.2.1 Motivation: Total Energy Consumption Dominated by Memory Access

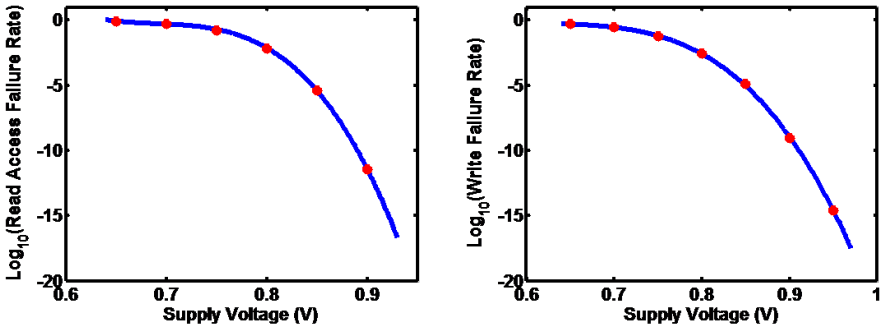
The overall energy consumption of NNs during inference/testing depends on the system architecture, frequency of operation, bit precision, memory type and size, etc. Usually memory access energy dominates the total energy consumption. Based on our energy models, we observed that in FCNs, memory access consumes 5.34x more energy than computation energy. However, in CNNs, dominance of memory access energy is comparably less (2.5x of computation energy) due to weight sharing in the convolutional layers.

Memory access power is dominated by the read and write accesses during inference/testing. We note that the number of synapses is typically two to three orders of magnitude greater than the number of neurons. Hence, the on-chip synaptic memory, conventionally designed using 6T SRAM, consumes significant amount of access and leakage energy.

3.2.2 Effects of Voltage Scaling on 6T SRAM

The supply voltage of 6T SRAM based synaptic memory can potentially be scaled to lower the energy consumption. However, 6T bitcells are susceptible to read-access and write failures at scaled voltages. The failures are aggravated in scaled technology nodes due to random process parameter variations [47–49]. The random variations effectively change the relative strength of the transistors constituting the individual bitcells. This negatively impacts the ability to read from (write into) a 6T bitcell within the stipulated time duration, resulting in read-access (write) failures.

Neural networks, being inherently resilient to small perturbations in the synaptic weights, enable the supply voltage of 6T SRAM based synaptic storage to be scaled moderately. This reduces the energy consumption for a negligible loss in the classification accuracy. We quantitatively analyzed the impact of supply voltage scaling on the stability of 6T SRAM. First, we designed a robust 6T bitcell in 45nm technology to operate at a nominal supply voltage of 0.9V and meet the static noise margin specifications in [30]. The 6T bitcells in a 128×512 sub-array were then subjected to random threshold voltage fluctuations to mimic the process parameter variations. We note that the optimal sub-array configuration was obtained from CACTI [50] for a memory size of 8KB. Finally, we performed Monte Carlo SPICE simulations to estimate the read and write failures at different supply voltages.



(a) (b)

Fig. 3.9.: (a) Read access failure rate versus supply voltage and (b) Write failure rate versus supply voltage, for a 6T bitcell.

3.2.3 8T-6T Hybrid Memory

Fig. 3.9 shows that 6T bitcell failures increase exponentially as the supply voltage is scaled. Therefore, aggressive voltage scaling of 6T SRAM could potentially result in unacceptable degradation in accuracy due to corruption of a large fraction of the synaptic weights. Reference [30] explored a hybrid 8T-6T SRAM [21], where few

Most Significant Bits (MSBs) of the synaptic weights are protected in reliable 8T bitcells while the relatively tolerant Least Significant Bits (LSBs) are stored in 6T bitcells. The enhanced stability of an 8T bitcell at scaled voltages can principally be attributed to decoupled read and write paths, leading to independent optimization for the respective operations (read and write). Monte Carlo analysis of a similarly sized sub-array of 8T bitcells yielded negligible failures in the voltage range of interest. Therefore, a hybrid 8T-6T SRAM allows the supply voltage to be scaled aggressively, which leads to improved energy efficiency at the expense of an added area penalty. Hence, it is important to minimize the number of 8T bitcells in order to achieve the best trade-off between classification accuracy, area, and energy consumption.

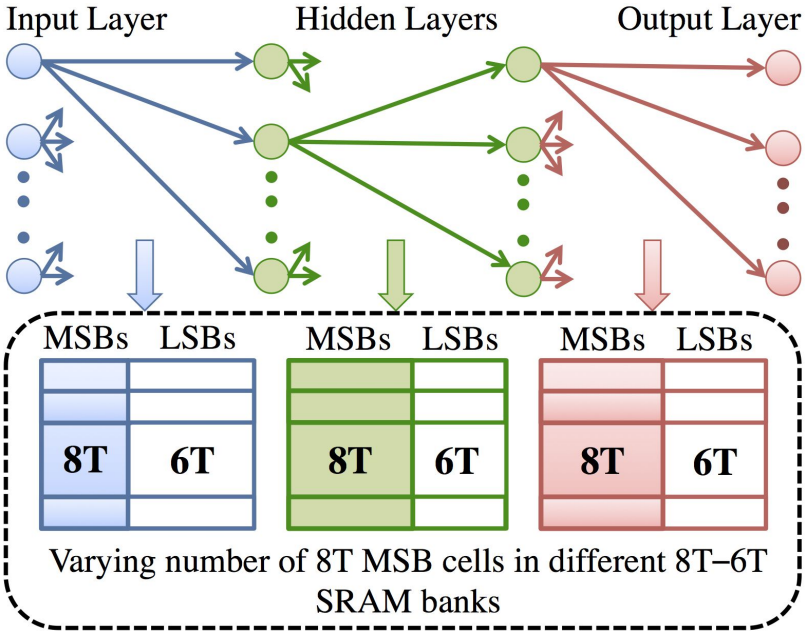


Fig. 3.10.: Synaptic sensitivity driven hybrid 8T-6T memory architecture [30].

For our analysis, we use the synaptic sensitivity driven hybrid memory architecture proposed in [30] and illustrated in Fig. 3.10. It consists of multiple 8T-6T SRAM banks, each of which stores the strength (weight) of synapses interconnecting a pair of neural network layers. The number of 8T MSB cells in each bank is chosen based on the sensitivity of the corresponding synapses to perturbation. For deep FCNs, we

find that the synapses interconnecting the input and the initial hidden layers have a bigger impact on accuracy, and hence must use a higher number of 8T MSB cells. Additionally, the synapses feeding into the output layer are significant since they directly influence the classification performance. For deep CNNs, the convolutional weight kernels ought to be stored in hybrid banks with greater number of 8T MSB cells.

3.2.4 Design Technique and Realization

For the voltage-scaled approximate memory, the constituent 6T and 8T bitcells were designed and subjected to Monte Carlo SPICE simulations to determine the read-access and write failure probabilities at reduced voltages. The energy consumption of the hybrid 8T-6T SRAM bank including the peripherals is obtained using CACTI [50] for the 45nm process technology.

At the system-level, the deep learning toolbox [45] and MatConvNet [46], which are MATLAB based open source neural network simulators, are used to model the approximations and evaluate the performance (classification accuracy) of the DNNs under consideration. We implemented the fully connected and convolutional NNs without data augmentation, batch normalization, and dropout features to primarily single out the effects of different approximations. Details of the benchmarks used in our experiments are listed in Table 3.2.

3.2.5 Results

We scaled down the supply voltage of 6T SRAM based synaptic memory to achieve energy efficiency by exploiting the intrinsic error resiliency of neural networks. Our simulations on a deep FCN ([784 1200 600 10]) trained for MNIST digit recognition showed that the supply voltage could be lowered till 0.85V from the nominal voltage of 0.90V for negligible accuracy loss. Further reduction in voltage resulted in substan-

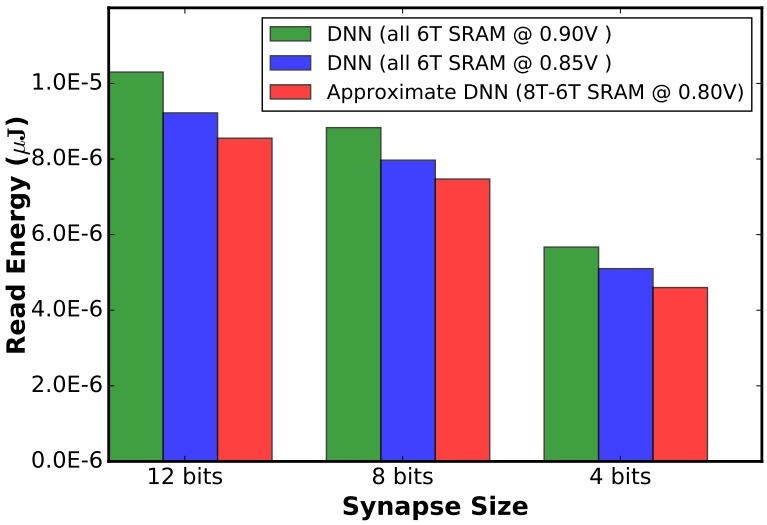
tial accuracy degradation. Therefore, we used the synaptic sensitivity driven hybrid memory architecture for aggressive voltage scaling.

We explored three different hybrid memory configurations corresponding to a synaptic precision of 12 bits, 8 bits, and 4 bits respectively. Each memory configuration consists of three 8T-6T SRAM banks to store the weight of synapses interconnecting every pair of layers of the deep FCN under investigation. The number of 8T MSB cells in each bank (shown in Table 3.3) is determined based on synaptic sensitivity as described in Section 3.2.3.

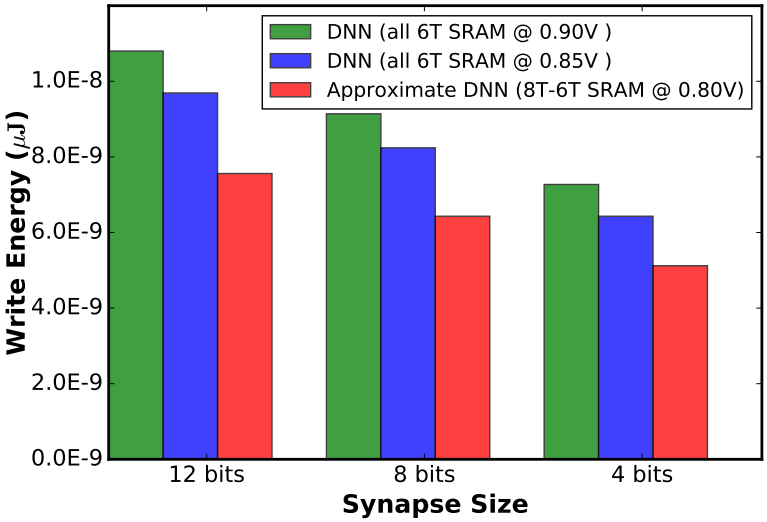
Table 3.3.: Synaptic sensitivity driven hybrid memory configuration

Synaptic Precision	#8T MSBs in Bank-1	#8T MSBs in Bank-2	#8T MSBs in Bank-3
12 bits	4	3	5
8 bits	3	2	3
4 bits	1	1	2

Our simulations indicate that the hybrid 8T-6T SRAM enable the voltage to be scaled down to 0.80V, which offers improved memory access energy efficiency compared to an all-6T SRAM that needs to be operated at 0.85V for negligible accuracy degradation. This is corroborated by figure 3.11, which shows that the deep FCN under consideration using the hybrid 8T-6T SRAM (operating at 0.80V) consumes lower memory access energy relative to an all-6T SRAM (operating at 0.85V). However, the improvement in energy consumption decreases as the synaptic precision is lowered from 12 bits to 4 bits. This can be attributed to a reduction in the complexity of the memory peripheral (decoding and sensing) circuitry. The reduced complexity minimizes the improvement in memory access energy achieved by operating at 0.80V relative to that expended at 0.85V. We note that deep CNNs (trained on CIFAR-10) also demonstrate a similar trend.



(a)



(b)

Fig. 3.11.: Comparison of total memory (a) read energy and (b) write energy, for classification of one image under iso-accuracy condition. Accuracy for 12, 8 and 4 bit networks are ∼98.9%, ∼98.8% and ∼98.5%, respectively with degradation up to 0.5% for 12 bit and 8 bit synapse, and ∼1.25% for 4 bit synapse.

Note that this approximation is non-deterministic, as a result of which retraining cannot be used to retrieve portion of the lost accuracy. Therefore, we scale the voltage down only till that point where the accuracy degradation is marginal ($<$0.5% for 12 bit and 8 bit synapse, and $\sim$1.25% for 4 bit synapse), for a fair iso-accuracy comparison with the conventional DNN. We additionally found that the accuracy could occasionally improve ($<$1%) even after applying memory approximation due to the non-deterministic nature of the approximation.

4. ALGORITHMIC LEVEL APPROXIMATIONS

In this chapter, we consider two algorithm level approximations, *viz.* lower complexity networks and pruning.

4.1 Prunning of Synapses

Pruning has been shown to reduce the complexity of an NN tremendously [32, 33]. Essentially, the insignificant synaptic connections are eliminated (or pruned) to achieve an order of magnitude reduction in the NN computations with minimal impact on accuracy. Fig. 4.1 shows the distribution of the synaptic weights for a deep FCN (with 784 input neurons, 2 hidden layers consisting of 1200, 600 neurons and 10 output neurons denoted as [784 1200 600 10]) trained on the MNIST dataset using a precision of 12 bits. We observed that a substantial portion of the weights carry very small values (experimentally determined to be below 0.04 for this application). Based on our analysis, we found that these weights are insignificant and can be removed (*pruned*) without having a significant impact on the accuracy. We observed that 8-bit synapses follow a similar weight distribution as 12 bit synapses. Hence, pruning the 8 bit synapses (below 0.04) also had negligible impact on accuracy. For an NN with 4 bit synapses, the lower bit precision naturally truncates a majority of synaptic weight values to 0. However, a synaptic precision of 4 bits, although more computationally efficient than 8 or 12 bits, is not preferable, since it drastically degrades the accuracy. The energy/accuracy trade-offs offered by NNs with different synaptic bit-precisions will be discussed in subsection 4.1.2.

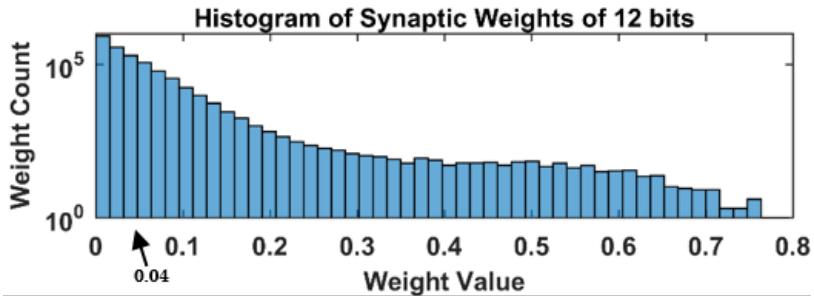


Fig. 4.1.: Synaptic weight distribution of a Deep FCN trained on MNIST: [784 1200 600 10] with 12 bit synaptic weights.

4.1.1 Prunning Methodology

Next, we describe the three-step process used to prune the synaptic weights. First, we train the network to learn all the synaptic connections. Then, we remove (or prune) the unimportant connections based on a pruning threshold that is determined from the distribution of the learned synaptic weights. Finally, we retrain the network to adjust the weights of the remaining connections to reclaim a significant portion of the accuracy lost due to pruning. The percentage of the synaptic weights that can be pruned is estimated by analyzing its impact on the network accuracy. Our analysis (Fig. 4.2) indicates that almost 80% of the synaptic connections of a trained deep FCN (for MNIST) can be pruned for negligible accuracy degradation.

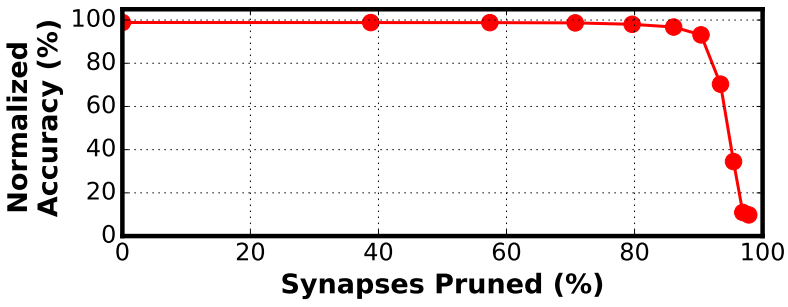
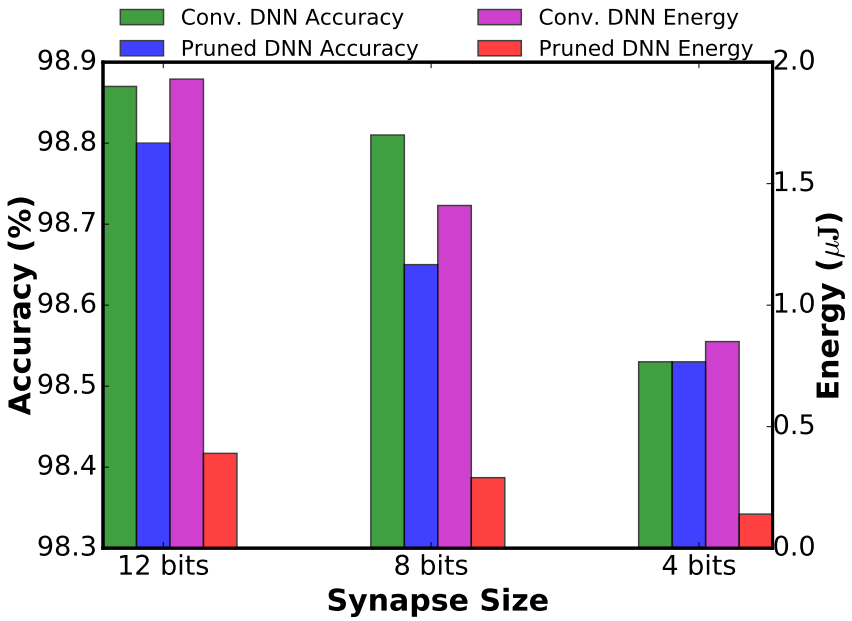


Fig. 4.2.: Effect of pruning on accuracy for a Deep FCN trained on MNIST: [784 1200 600 10] with 12 bit synaptic weights.

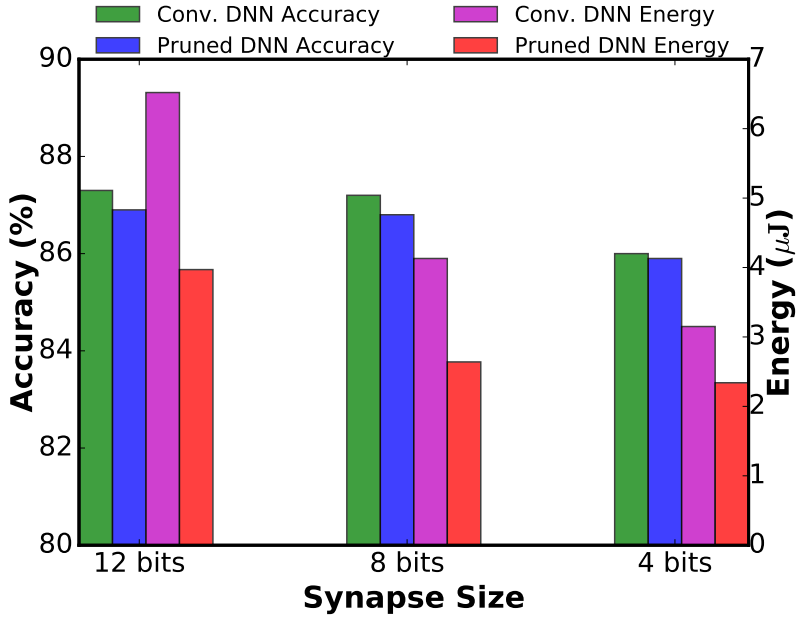
4.1.2 Results

We systematically pruned the synaptic weights using the procedure described in sub-section 4.1.1. The energy-accuracy comparison between conventional DNN and pruned DNNs is shown in Fig. 4.3. Our analysis on a deep FCN trained on the MNIST dataset using 8-bit and 12-bit synapses showed that ∼80% of the synaptic connections could be removed with minimal accuracy degradation, which leads to roughly 80% savings in energy compared to the conventional (un-approximated) DNNs. We further note that FCNs with 4-bit synapses allowed up to 84% of the connections to be pruned, resulting in larger energy benefits. Fig. 4.3a shows that the accuracy degradation is less than 0.20% for pruned DNNs across different bit-width configurations.

A similar analysis on a deep CNN trained on the CIFAR-10 dataset indicated that 41.24%, 39.51% and 25.75% of the synaptic connections could be eliminated for 12-bit, 8-bit and 4-bit synapses, respectively, with negligible accuracy degradation. It can be seen from Fig. 4.3 that pruned networks consistently provide better energy consumption than conventional DNNs.



(a)



(b)

Fig. 4.3.: Energy/accuracy trade-off comparison, between pruned DNNs and conventional DNNs, is shown for (a) MNIST trained on Deep FCN and (b) CIFAR10 trained on Deep CNN, for different synapse sizes.

4.2 Low Complexity Networks

For a given baseline DNN, we train a low complexity network (containing lesser number of layers and/or neurons per layer) in order to achieve a significant reduction in energy for a small reduction in accuracy. There are two basic approaches that can be used to derive the lower complexity networks: i) reduce the number of hidden/convolutional layers, or, ii) reduce the number of neurons/convolutional kernels in the hidden/convolutional layers. We employ these network approximations and thereafter evaluate the DNN's classification accuracy. If the achieved accuracy is above the target accuracy, then the network is further reduced and retrained. If the network is slightly less than or equal to the target accuracy, then retraining is terminated. Finally, the network is optimized by bit-precision scaling as explained below. During bit-precision scaling, the input and synaptic bit width is reduced until the point where accuracy starts to degrade so as to determine the optimal bit precision for the network. The training methodology is depicted in Fig. 4.4.

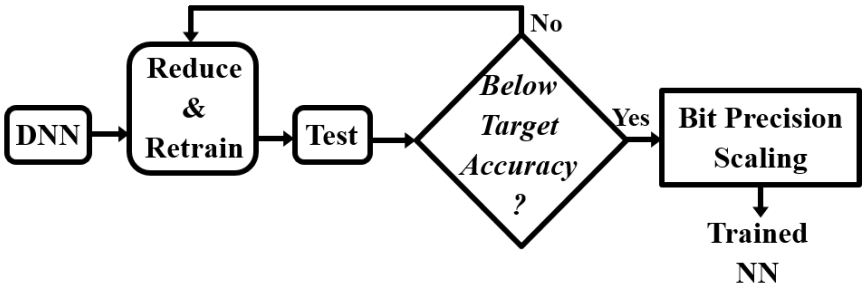


Fig. 4.4.: Overview of the NN training methodology for employing network approximations.

4.2.1 Results

We evaluated lower complexity networks that were additionally precision-scaled to reduce energy for a given accuracy. In this sub-section, we present results for lower complexity NNs for the datasets under investigation.

First, we investigate low complexity FCNs, trained on the MNIST dataset, consisting of a single hidden layer. We started with twice number of neurons in the only hidden layer of the low complexity network compared to the number of neurons in the first hidden layer of the deep FCN. The number of neurons in the hidden layer of the low complexity network is decreased until the desired accuracy is achieved. Bit-precision scaling is subsequently applied on the low complexity FCN to determine the optimal synapse bit-width. For a baseline deep FCN [784 1200 600 10] trained on MNIST, a few low complexity FCNs are listed in Table 4.1.

Table 4.1.: Low Complexity FCNs Trained on MNIST

#Hidden Layer Neuron	Bit Precision	Accuracy of NN (%)	Computation Energy Consumption (J)	Energy Savings (%)
2400	9	98.88	1.82	5.35
2400	8	98.85	1.91	0.73
2100	8	98.85	1.40	27.07
1800	9	98.77	1.36	29.01
1550	9	98.73	1.17	38.87
800	9	98.64	0.61	68.45
650	12	98.52	0.60	69.00
425	7	98.36	0.25	86.80
400	10	98.22	0.32	83.23

Savings are computed by considering the conventional DNN (12bit) as standard.

Next, we explored low complexity CNNs on the CIFAR10 dataset, where we likewise begin with a single convolutional layer, and decrease the number of weight kernels to attain desired accuracy. Our analysis showed that such a shallow (only one MLPConv block) CNN failed to match the accuracy of a deep CNN (87.3%) even with greatly increased number of kernels, with 78% being the maximum achievable accuracy. Therefore, we utilized two MLPConv blocks with equal to or lesser number of kernels in each than the Deep CNN, to realize the low complexity CNNs. This enabled the low complexity CNN to attain an accuracy of 86% for a precision of 8 bits and above, while the deep CNNs consistently delivered an accuracy of greater than 87% for equivalent synaptic bit-widths. For a baseline deep CNN [1024×3 $(5 \times 5)192c$ $160fc$ $96fc$ $(3 \times 3)mp$ $(5 \times 5)192c$ $192fc$ $192fc$ $(3 \times 3)mp$ $(3 \times 3)192c$ $192fc$ $10o$] trained on CIFAR10, a few low complexity CNNs are listed in Table 4.2.

Table 4.2.: Low Complexity CNNs Trained on CIFAR10

#Neurons	#Synapses	Accuracy of NN (%)	Computation Energy Consumption (J)	Energy Savings (%)
542762	538719	86.0	2.25	65.49
405546	379281	85.6	1.6	75.46
359466	324768	85.3	1.36	79.14
268330	223374	83.6	0.94	85.58
221226	171522	80.5	0.70	89.26

Savings are computed by considering the conventional DNN (12bit) as standard.

5. COMBINATION OF APPROXIMATE TECHNIQUES

5.1 Optimized Baseline Deep Neural Networks

Optimized DNNs are bit-precision scaled DNNs without any approximations. We identify the DNN architecture that provides the best classification accuracy for a given synaptic bit-precision. The DNN configuration thus determined is chosen as the optimized network, which is then subjected to different approximations presented in this work. We perform the analysis on deep FCNs trained on the MNIST digit recognition dataset, and deep CNNs trained on the CIFAR-10 image recognition dataset. Fig. 5.1 demonstrates the effect of bit precision scaling. Our experiments indicate that a deep FCN (with 784 input neurons, 2 hidden layers consisting of 1200, 600 neurons and 10 output neurons, denoted as [784 1200 600 10]), offers the best accuracy of 98.87% on the MNIST dataset for synaptic precision of 12 bits. The accuracy degradation was found to be minimal (<0.5%) up to precision of 4 bits. For the CIFAR-10 dataset, we implemented a CNN with 3 MLPConv [51] (combination of convolutional and fully connected layers) blocks: [1024×3 $(5\times5)192c$ $160fc$ $96fc$ $(3\times3)mp$ $(5\times5)192c$ $192fc$ $192fc$ $(3\times3)mp$ $(3\times3)192c$ $192fc$ $10o$]. The input layer is $32\times32\times3$. All convolutional layers use 5×5 kernel size with different number of feature maps. A 3×3 max pooling window is used after each MLPConv block. The final features from the last block are then fully-connected to a 10-neuron output layer. This Deep CNN provided the maximum accuracy of 87.3%. There was reasonable deterioration in the accuracy (<1.5%) for a precision of 4 bits.

Researchers have also shown networks with 1 bit synapse only, termed Binary networks [52]. The 4 bit networks use 4 times the memory for each synapse and needs multipliers for neuronal operation compared to binary networks where multiplication

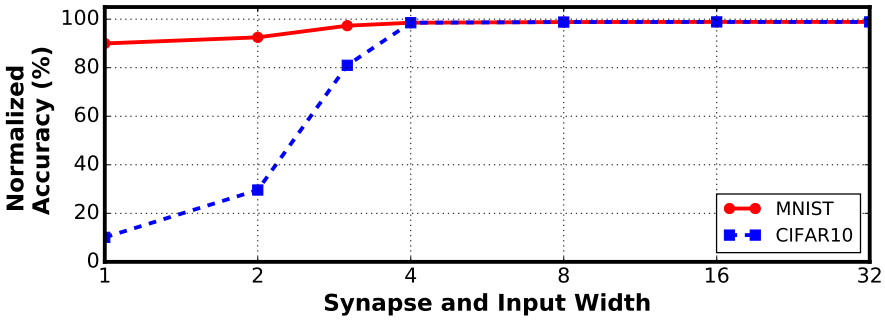


Fig. 5.1.: Effect of bit precision scaling (with retraining) on benchmark applications.

can be replaced with simple bitwise operations. However, such networks are much larger compared to the networks used in this work.

5.2 Approximate Multiplier, Pruning and Approximate Memory

While each approximation technique, discussed earlier, independently provides substantial energy benefits for a DNN, we combined the techniques into a synergistic framework as illustrated in Fig. 5.2 to maximize the energy savings. First, we ***prune*** the insignificant synaptic weights from the trained network. Next, we introduce appropriate weight restrictions as necessitated by the ***approximate multipliers***. Then, we ***retrain*** the network to minimize the accuracy loss suffered due to the aforementioned approximations. During retraining, only the non-zero weights are updated while accounting for the weight constraints imposed by the approximate multiplier topology.

Finally, we introduce bit-flips in the resultant synaptic weights to incorporate the read-access and write failures of the voltage-scaled ***approximate memory***, and estimate the network accuracy. Note that the memory failures are distributed randomly. Therefore, retraining the network further will not help in regaining any fraction of the accuracy lost due to voltage scaling. We would like to point out that Lower complexity NN is not considered while combining different approximate techniques. This is because, once this approximation is applied to a DNN, it becomes

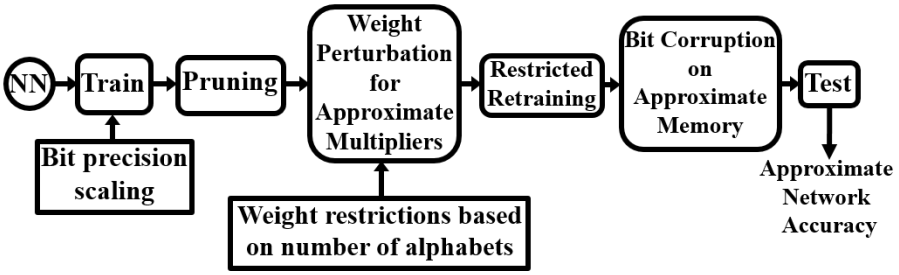


Fig. 5.2.: Flow diagram of the proposed combined approximation process of a NN.

a shallower network with its 'degrees of freedom'/flexibility diminished. These lower complexity networks are very sensitive to weight perturbation since they have very small number of learning parameters and therefore cannot cope with further hardware approximations without significant accuracy degradation. Therefore, we consider lower complexity networks separately from the remaining approximations.

5.2.1 Retraining to Mitigate Accuracy Loss

We had noted in chapter 3 and 4 that retraining is used to mitigate the accuracy degradation incurred due to certain approximations including pruning and approximate multiplication. We now highlight the importance of adapting the learning rate for efficiently retraining the network with these approximations in place. The learning rate, which essentially is a multiplication factor in the weight update equation [45], influences the speed and quality of learning. It needs to be precisely regulated while using the approximate multiplier due to the non-uniformity in the distance between the allowed weight levels which is explained in sub-section 3.1.5. Retraining also improves accuracy which was lost due to the application of synaptic pruning. Hence, it is necessary to utilize the optimal learning rate for retraining an approximate DNN. Fig. 5.3 illustrates the flow diagram of the retraining process, which begins with the highest learning rate that was used to originally train the DNN without approximation. If the accuracy improves, retraining is carried on with the same learning rate

for few more iterations, until there is no significant change in the accuracy. On the other hand, if the accuracy does not improve, the learning rate is regulated (reduced by a factor) and the approximate DNN is further retrained. This process of regulating the learning rate is continued until the accuracy improvement saturates. The initial high learning rate during retraining allows the synapses to compensate for the weight perturbations despite the weight constraints still being applied. However, high learning rate incurs oscillations and may not allow the retraining to converge to the optimal solution. Therefore, learning rate is reduced based on the retrained network performance. The low learning rate helps to dampen the oscillations and allow the retraining to converge. We note that the retraining overhead for an approximate DNN is negligible compared to the number of epochs required to train the original DNN (without approximations). Also, retraining does not affect the energy consumption during inference/testing.

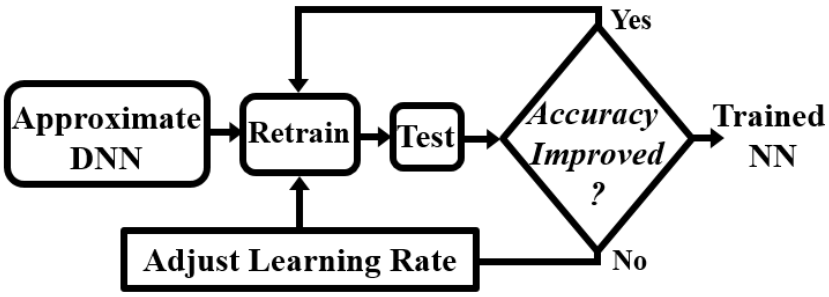


Fig. 5.3.: Flow diagram of the proposed combined approximation process of a NN.

5.2.2 Results

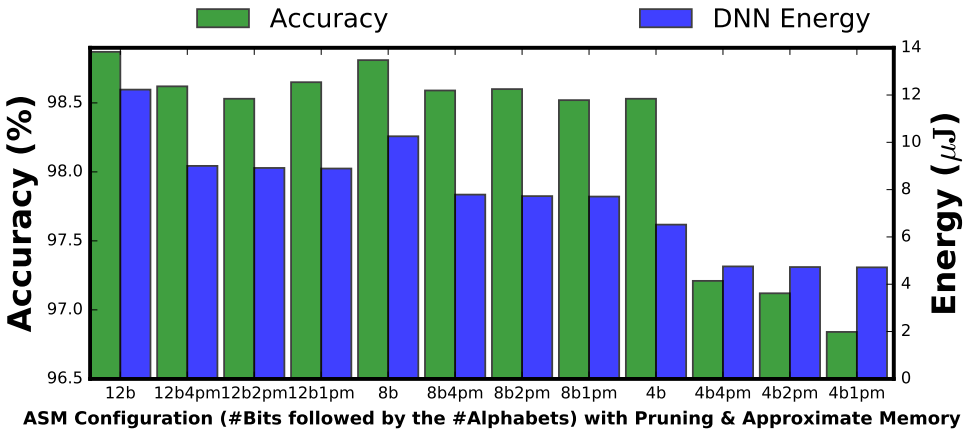
Our results thus far indicate that approximate DNNs demonstrate improved energy efficiency compared to conventional DNNs both in terms of computation and memory-access energy with minimal accuracy degradation. We then combined all the approximation techniques together using the steps outlined in sub-section 5.2.1. We found that, the all-inclusive approximate DNNs incurred greater accuracy loss

as expected. The memory approximation was the dominant factor for the accuracy degradation since the accuracy lost due to voltage scaling could not be regained by retraining the network. Approximate memory also gave rise to a good amount of non-zero weight values which were previously pruned. This in turns reduced the energy savings achieved in computation by pruning. Even with these setbacks, the combined network provided good amount of energy savings both in computation and memory access. This is illustrated in Fig. 5.4a, which shows that approximate DNNs (FCN trained on MNIST) achieve significant amount of energy savings with synapse pruning, approximate multiplication and approximate memory being applied simultaneously. They provide up to 7.85x improvement in the computation energy over conventional DNNs of equivalent bit precision. Approximate Deep CNNs (trained on CIFAR-10) also indicate a similar trend (Fig. 5.4b), and offer up to 2.76x reduction in the computation energy consumption over conventional DNNs of equivalent bit precision. However, in Fig. 5.4a, all the energy consumption bars for the approximate networks look similar as all of them have same memory access energy (for their respective bit-precision), which dominates the total energy consumption. On the other hand, in Fig. 5.4b, the energy consumption bars for the approximate networks are distinct since the memory access energy is less dominant in CNNs. Both Approximate DNNs (FCN and CNN) provide up to 19% read energy and 30% write energy savings compared to DNNs operated at nominal voltage (Fig. 3.11).

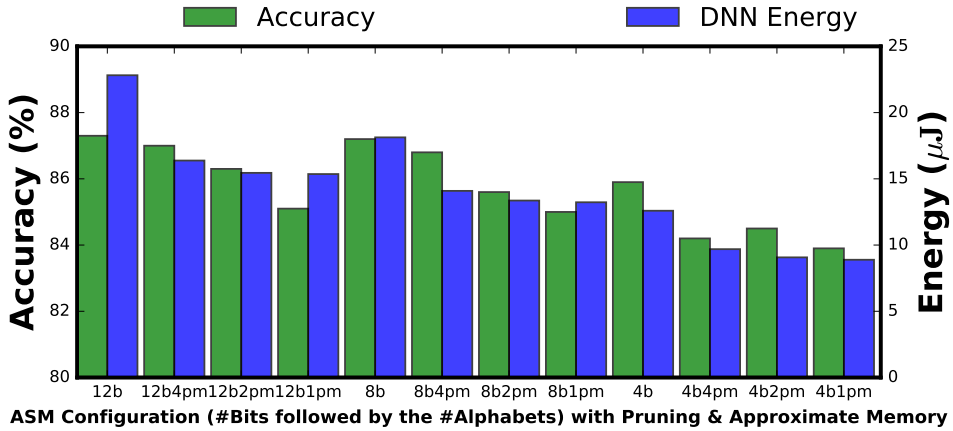
However, it is not necessary to combine all the approximation techniques. Based on the energy-quality requirements, one or more approximation techniques can be applied. We experimented with different combinations of the three approximation techniques. The results are discussed in the following sub-section.

5.3 Comparing Approximate Networks

Approximations cause accuracy degradations and it has an inverse relationship with energy savings. The extent of accuracy degradation is mostly dependent on



(a)



(b)

Fig. 5.4.: Energy/accuracy trade-off comparison between Approximate DNNs and Conventional DNNs, for (a) MNIST dataset on FCN and (b) CIFAR 10 dataset on CNN, where pruning of synapses, approximate multipliers and approximate memory are used simultaneously. 12b: 12 bit synapse NN, 12b4pm: 12 bit synapse NN with 4 alphabet ASM with pruning and approximate memory. Similar notations for 8 bit and 4 bit NNs.

the intensity of approximations. At the same time bit-precision of the neurons and synapses therefore plays an important part in controlling the energy-accuracy trade-

off. Different approximations provide varied energy-accuracy trade-offs. Depending on the energy-accuracy specifications for a given application, different approximations need to be employed. So far we have investigated 4 different kind of approximations at two different levels of NN realization (algorithm and hardware). In this sub-section, we will compare these approximations in an attempt to find out maximal beneficial approximation method.

Fig. 5.5 shows energy-accuracy trade-off comparison between Conventional DNNs and different approximate DNNs listed below:

i) Conventional DNN (un-approximated baseline)

ii) Pruned DNN (algorithm level approximation)

iii) Approximate Memory based DNN (hardware level approximation)

iv) Approximate Multiplier based DNN (hardware level approximation)

v) Low Complexity Network (algorithm level approximation)

vi) DNN with Approximate Multiplier and Memory

vii) DNN with Approximate Multiplier and Pruning

viii) DNN with Approximate Memory and Pruning

ix) Combined Approximated DNN (pruning, approximate multiplication and memory)

From Fig. 5.5a, we observe that the low complexity networks cannot match the higher baseline accuracy. For high accuracy requirements, pruning and approximate multiplier based DNNs provide better energy efficiency compared to low complexity networks. For low accuracy requirements, low complexity networks are better candidates, if we consider only computation energy consumption. Such substantial deterioration in the accuracy of low complexity networks is not acceptable for practical purposes. Moreover, in Fig. 5.5a, we observe that memory approximated networks provide better overall energy-efficiency compared to low complexity networks. We focus mostly on high accuracy regime of the energy-accuracy graph and do not consider low complexity networks.

Approximate multiplier based DNNs provide good amount of computation energy savings with minimal accuracy loss. Note, pruned DNNs provide better computation energy efficiency than all other approximation techniques. Pruning of synapses along with approximate multipliers can be used to achieve greater savings in computational energy for less than 0.15% drop in accuracy. Note that the order of the approximations (pruning after or before applying weight restrictions for approximate multiplier) does not matter, since retraining is applied after employing both approximations. On the other hand, Approximate memory based DNNs consume similar computation energy as conventional DNNs with slightly lower classification accuracy. However, they provide good amount of memory energy savings as discussed in section 3.

The combination of 3 approximation techniques (pruning, approximate multiplier and memory) provides slightly lower computational energy efficiency than the combination of pruned and approximate multiplier based network. This can be attributed to the fact that the memory approximation introduces bit corruption, which converts some of the pruned weights to non-zero values, and thus reduces the computation energy savings. However, memory approximation individually offer large amount of savings in memory access energy as total energy consumption for a deep FCN is greatly dominated by memory accesses.

In Fig. 5.5b, a similar trend can be observed in the case of deep CNN trained on CIFAR10 dataset. One major difference is the comparatively lesser domination of memory access energy on the total energy consumption due to weight sharing in the convolutional kernels. Another distinctive feature is the less effectiveness of pruning as most of the kernel values are non-zero.

5.4 Comparison with other Low Power DNNs

There are several other works that focused on low power NN hardware architectures [27, 31, 53, 54]. In [27], resilient neurons are identified during training, then precisions (bit-widths) of the input operands and the synapse weights corresponding

to those neurons are modulated based on their degree of resilience. The network is retrained to compensate for the weight perturbations. The approximations applied on the network is non-uniform as different neurons/synapses have different bit-precision.

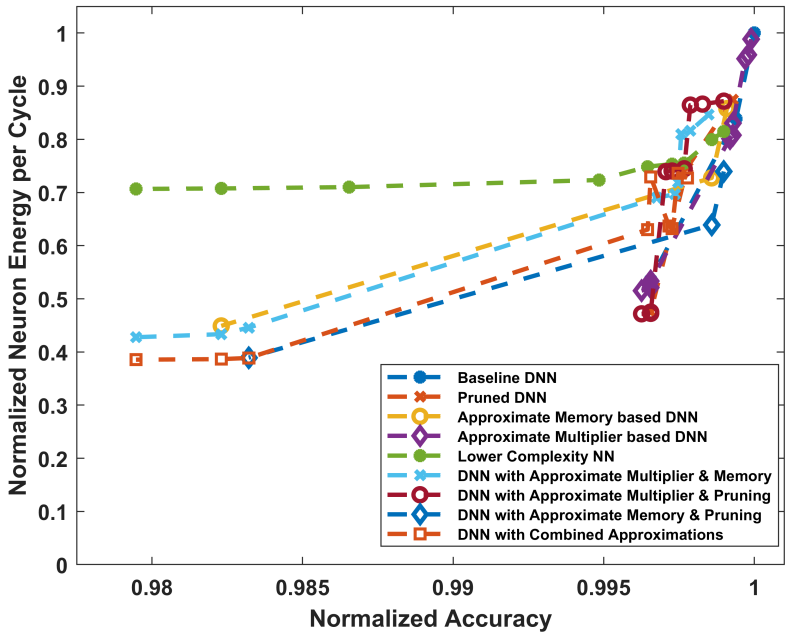
In [31], resilient synapses are identified and are processed with reduced bit-precision. Then computations with those synapses are done in an approximate processing engine, containing approximate multiplier. On the contrary, in our work we apply uniform approximation throughout the network. The approach [31] achieves ∼63.6% computation energy savings with a shallow FCN ([784 144 100] for MNIST dataset), while incurring ∼1.45% accuracy degradation compared to our shallow FCN ([784 120 100] for MNIST dataset) which achieves ∼50% computation energy savings with ∼1.34% accuracy degradation. Note however, the baseline in [31] is a 32 bit network, while in our case the baseline is a 12 bit network.

Authors in [53] proposed a new framework that enables aggressive voltage scaling for DNN accelerators without compromising classification accuracy. Aggressive voltage scaling can cause timing errors. The authors in [53] show that with dynamic voltage scaling and zero-skipping (pruning), they are able to achieve up to 89% computation energy savings with ∼1% accuracy degradation for MNIST trained on a 4 layer FCN. In our work, we achieved best results using pruning and 4 bit 1 alphabet based multiplier for training a 3 layer FCN with MNIST dataset. Using this network, we achieved ∼95% computation energy savings with only ∼0.30% accuracy degradation. Moreover, authors in [53] showed that maximum energy savings comes from zero-skipping (pruning) which corroborates our results with synaptic pruning.

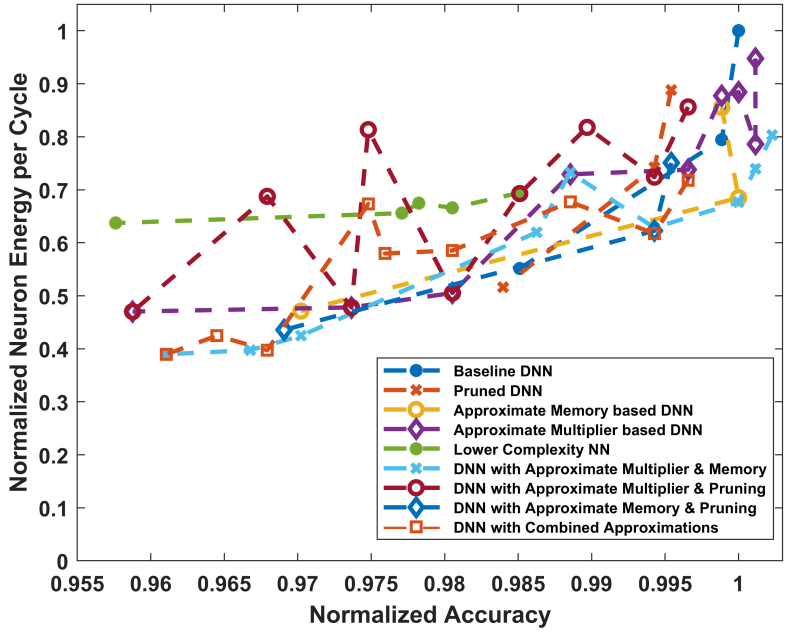
Authors in [54] proposed the use of voltage scaled dot-product-engines (DPE) units, which operate in the near threshold voltage (NTV) regime. To mitigate the timing error occurring due to the reduced operating voltage, they also proposed a new statistical error correction (SEC) technique referred to as rank decomposed SEC (RD-SEC). This technique is particularly well suited for low-cost error detection and correction. This work showed that processors for DNNs can be operated at very low voltage with these techniques beings appropriately applied. However, in [54], we

were unable to find necessary information (computation energy savings, classification accuracy) which are required for a quantitative comparison.

Researchers have also shown networks with 1 bit synapses, termed Binary networks [55, 56]. However, such networks are much larger in size compared to the networks used in this work. Another important point to be noted here is that we apply bit precision scaling to network inputs to take advantage of lower precision MAC units. However, in binary networks, only synapses and activations are quantized. Authors in [55] achieved ∼99% accuracy with a deep FCN ([784 4096 4096 4096 100] for MNIST dataset) using binary synapses, compared to our smaller FCN ([784 1200 600 100] for MNIST dataset) which achieves ∼98.50% accuracy with 4 bit 1 alphabet based multiplier (4 bit synapses) and pruning. Also in [55], for CIFAR10 dataset, ∼89.9% accuracy was achieved with a very deep CNN (VGG [54]), compared to our simpler CNN (NIN [47]) which achieves ∼86% accuracy with 8 bit 2 alphabet based multiplier and pruning. Another work [56] employed approximate hardware (multiplier and adder) in binary convolutional networks. The authors in [56] reported accuracy of 98.37% for MNIST trained on LeNet5 and 84.87% for CIFAR10 trained on VGG16 [38], which are less than the accuracies achieved by our proposed optimum network configurations. Note, the networks are much larger compared to our networks. For example, VGG16 has ∼10x more parameters and requires ∼12.5x more computations than NIN architecture (deduced from [57]).



(a)



(b)

Fig. 5.5.: Energy/accuracy trade-off comparison between different approximate DNNs and Conventional DNNs, for (a) MNIST dataset on FCN and (b) CIFAR 10 dataset on CNN. Here baseline for normalization is a 12 bit un-approximated DNN.

6. GABOR FILTER ASSISTED FAST AND EFFICIENT LEARNING FOR CNN

One of the major challenges for convolutional networks is the computational complexity and time needed to train large networks. Training of CNNs requires state-of-the-art accelerators like GPUs for extensive applications [58]. The large training overhead has restricted the usage of CNNs to clouds and servers. However, an emerging trend in IoT promises to bring the expertise of CNNs (image recognition and classification) to mobile devices that may lack or have only intermittent online connectivity. To ensure applicability of CNNs on mobile devices and widen the range of its applications, the training complexity must be reduced. Good amount of work has been done on speeding-up the training process through parallel processing [59], but not much work is found on improving the energy efficiency of CNN training. CNNs are trained using the standard back-propagation rule with slight modification to account for the convolutional operators [45]. The main power hungry steps of CNN training (back-propagation) are gradient computation and weight update of the convolutional and fully connected layers. In our proposed training, we achieve energy efficiency by eliminating a large portion of the gradient computation and weight update operations, with minimal loss of accuracy or output quality.

Classification accuracy is a primary concern for researchers in the machine-learning community. Different pre-processing models such as filters or feature detectors have been employed to improve the accuracy of CNNs. In fact, recent works employed Gabor filtering as a pre-processing step for training neural networks in pattern recognition applications [60,61]. Based on the human visual system, these filters are found to be remarkably appropriate for texture representation and discrimination. In [62, 63], the authors have attempted to get rid of the pre-processing overhead by introducing Gabor filters in the 1^{st} convolutional layer of a CNN. In [62], Gabor filters replace the

random filter kernels in the 1st convolutional layer. The training is then limited to the remaining layers of the CNN. The focus of such work was accuracy improvement. However, these approaches may also lead to energy savings. In [63], the Gabor kernels in the 1st layer were fine-tuned with training. In other words, the authors used Gabor filters as a good starting point for training the classifiers, which helps in convergence. In this work, we build upon the above works to propose a balanced CNN configuration, where fixed Gabor filters are not only in the 1st convolutional layer, but also in the latter layers of the deep CNN in conjunction with regular trainable convolutional kernels. Using Gabor filters as fixed kernels, we eliminate a significant fraction of the power-hungry components of the back-propagation training, thereby achieving considerable reduction in training energy. The inherent error resiliency of the networks allows us to employ proper blend of fixed Gabor kernels with trainable weight kernels to lower the compute effort while maintaining competitive output accuracy.

6.1 Gabor Filters

Many pattern analysis applications such as character recognition, object recognition, and tracking require spatially localized features for segmentation. Gabor filters are a popular tool for extracting these spatially localized spectral features [64]. Simple cells in the visual cortex of mammalian brains can be modeled by Gabor functions [65]. Frequency and orientation representations of Gabor filters are similar to those of the human visual system, and they have been found to be particularly appropriate for texture representation and discrimination. A particular advantage of Gabor filters is their degree of invariance to scale, rotation, and translation. In fact, [66] have shown that deep neural networks trained on images tend to learn first layer features resembling Gabor filters. This further corroborates our intuition of using pre-designed Gabor filters as weight kernels in a CNN configuration.

An appropriately designed Gabor filter extracts useful features corresponding to an input image. However, this would also require us to have separate uniquely designed

filters for every image that will be infeasible for large-scale problems. In order to have a generic approach, we have used a systematic method where filters are generated using a 'filter bank' [64–67]. The generated Gabor filters are then used to replace the weight kernels in the CNN configuration. We have used real values of 2D Gabor filters as weight kernels. Also, the filters used in our methodology are equally spaced in orientation (for instance, $\theta = 0°, 30°, 60°, 90°$ etc.) to capture maximum number of characteristic textural features.

6.2 Design Approach & Methodology

The use of fixed Gabor kernels, to exploit error resiliency of CNN, is the main concept of our proposed scheme. This section outlines the key steps of the proposed design methodology.

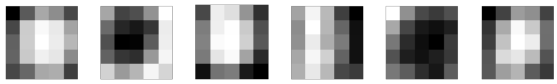
6.2.1 Energy Model for CNN Training

To realize the effect of Gabor kernels on energy, we developed an energy model to get the distribution of energy consumption for training the network. The energy model is based on the number of Multiply and Accumulate (MAC) operations in the training algorithm. The MAC circuits were designed in Verilog and mapped to 45 nm technology using Synopsys Design Compiler. Then the power and delay numbers from the Design Compiler were fed to the energy computation model to determine the distribution of energy consumption. From the energy distribution for training a CNN, we found that in a conventional CNN denoted by [784 6c 2s 12c 2s 10o] (2 convolutional layers (6c and 12c) each followed by a sub-sampling layer (2s), and finally a fully connected output layer (10o)), the 2^{nd} convolutional layer uses 27% of the overall energy consumption during training, while the 1^{st} convolutional layer consumes 20%. We realized that, in order to achieve energy efficiency, the energy spent on these layers needs to be minimized, and using Gabor filters as fixed kernels is the key.

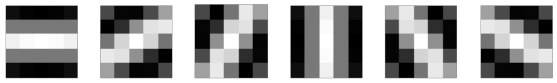
6.2.2 Gabor Filters as Fixed Convolutional Kernels

Gabor Filters in First Convolutional Layer

We designed a CNN, by replacing certain weight kernels with fixed Gabor filters. The network has 2 convolutional layers each followed by a sub-sampling layer, and finally a fully connected layer. The 1st convolutional layer has k kernels, while the 2nd convolutional layer has $2k$ kernels for each of the k feature maps of the 1st layer, resulting in total of $2k^2$ kernels in the 2nd layer. In this specific example, we consider k=6. Hence, for our example, the 1st convolutional layer consists of 6 kernels and the 2nd convolutional layer consists of 72 kernels. The 12 feature maps from the 2nd layer are used as feature vector inputs to the fully connected layer which produces the final classification result. We used 6 Gabor kernels (5×5 sized), which are equally spaced in orientation, (with $\theta = 0°, 30°, 60°, 90°$ and $150°$) to replace the regular kernels of the 1st convolutional layer. The network was trained on MNIST dataset [8]. The regular trainable kernels of the 1st layer of the CNN after 100 epochs and fixed Gabor kernels are shown in Fig. 6.1 to have a comparative view. The results of this configuration are shown in Table 6.1 (Row 2 corresponding to Fixed Gabor/Trainable CNN configuration).



(a)



(b)

Fig. 6.1.: (a) Trained kernels in 1st convolutional layer. (b) Fixed Gabor kernels equally spaced in orientation.

From Row 2 in Table 6.1, it can be clearly observed that the Fixed Gabor/Trainable CNN configuration has an accuracy comparable to the conventional CNN implementation with a marginal loss of 0.62% (baseline accuracy 99.09%). In addition, we observe 20.7% reduction in energy consumption, and 9.47% decrease in training time. The storage requirements for the networks trainable parameters remain unchanged.

Gabor Filters in Both Convolutional Layers

To achieve higher energy improvements, we turned the convolutional layers all Gabor, *i.e.* all the kernels in both the convolutional layers of the CNN were replaced with fixed Gabor filters. In the 2^{nd} convolutional layer, there are 12 kernels for each of the 6 output feature maps of the 1^{st} convolutional layer, in total 72 kernels. We used the same 6 Gabor kernels from sub-section 6.2.2, equally spaced in orientation, in the 1^{st} convolutional layer, and 12 Gabor kernels, equally spaced in orientation (with $\theta = 0°, 15°, 30°, 45°, 60°, 75°, 90°, 105°, 120°, 135°, 150°$, and $165°$), in the 2^{nd} convolutional layer. Considering the Gabor kernels as constants, 12 Gabor kernels are sufficient to replace the 72 kernels (convolved output of which are summed to produce 12 feature maps) of the 2^{nd} layer. In this case, same set of 12 Gabor kernels will convolve with each of the 6 output feature maps of 1^{st} layer in every training cycle. It is evident that the 6 kernels of the 1^{st} layer are also present in the 2^{nd} layer. This actually helps to carry the integrity of the previous layer to the following layer. The benefits observed from this Fixed Gabor/Fixed Gabor CNN configuration can be seen from Row 3 in Table 6.1. The training time reduced by 53%, leading to 1.93x improvement in energy consumption; also the storage requirement is drastically reduced. However, the big downside is the high accuracy degradation (5.85%), even for a simple dataset like MNIST. The configuration leads to intolerable accuracy degradation for more complex datasets.

Table 6.1.: Comparison between Different CNN Configurations

Configuration		**Accuracy Loss**	**Energy Savings**	**Training Time Reduction**	**Storage Savings**
1st Conv. Layer Kernels	**2nd Conv. Layer Kernels**				
Trainable	Trainable	–	–	–	–
Fixed Gabor	Trainable	0.62%	20.70%	9.47%	0%
Fixed Gabor	Fixed Gabor	5.85%	48.28%	53%	42.48%
Fixed Gabor	Half Fixed Gabor & Half Trainable	1.14%	34.49%	22.30%	23.15%

*Accuracy loss, energy savings, training time reduction and storage savings are computed by considering the conventional CNN results as baseline. The baseline accuracy is 99.09%.

Balanced Network Configuration for Maximum Benefits

As seen in sub-section 6.2.2, not training the 1st layer kernels incurs minor accuracy loss. However, we also observe that, fixing the 2nd layer reduces the accuracy drastically, while it gives very high energy savings. In order to avoid such accuracy degradation, we designed a blended CNN configuration where the 2nd convolutional layer is partly trained with a combination of fixed Gabor filter kernels and regular trainable weight kernels, while the 1st convolutional layer uses fixed Gabor kernels only. However, there can be multiple CNN configurations for different number of fixed and trainable kernels in the 2nd layer, each providing different amount of energy savings for corresponding accuracy degradation. To obtain the optimal configuration, we made an effort to balance the trade-off between accuracy and other parameters of interest, especially energy consumption. We conducted an experiment where we trained 5 configurations of blended CNN on MNIST dataset with i number of fixed Gabor kernels in the 2nd layer where i= 0, 3, 6, 9, 12. Each of the configurations was trained for 100 epochs. Fig. 6.2 shows the overall classification accuracy obtained

with the blended CNN configuration as we varied the number of trainable weight filters in the 2nd layer. Again, as in sub-section 6.2.2, Gabor kernels with equal spacing in orientation were used. It can be clearly observed that the accuracy degrades with the increase in the number of fixed Gabor filters. To select the optimal configuration, we imposed a constraint of maximum 1% degradation in the classification accuracy. In the plot (Fig. 6.2), the solid square corresponds to the accuracy for the Fixed Gabor/Trainable CNN configuration (Row 2 of Table 6.1) and the solid circle is the point where degradation in classification accuracy is 1%. This solid circle corresponds to a half-half configuration where half of the kernels in the 2nd layer is fixed while the other half is trainable. Fig. 6.2 also shows the energy savings, training time reduction and storage requirement savings observed with different configurations. As expected, the benefits (energy, training time, storage) increase as we fix more filters, beyond the half-half point. However, the decrease in accuracy is significant. We observed similar trend for CNNs trained on FaceDet and TiCH [68] datasets as well for this LeNet architecture. Based on this, we used the half-half configuration to implement the CNN for MNIST, FaceDet and TiCH, in order to get maximum benefits with minimal degradation in accuracy. The results are described in the following section.

The half-half balanced configuration implies that we have 6 fixed and 6 trainable kernels (in the 2nd layer) for each of the 6 feature maps from the 1st layer. Thus, total trainable kernels are 36 (6×6). Since the other 36 (6×6) kernels, for each of the 6 feature maps from the 1st layer, are fixed, they can be replaced by 6 Gabor kernels instead of 36. Again, the 6 fixed Gabor kernels used for the 2nd layer are same as the 6 Gabor kernels of 1st layer. Using the same kernels across the 1st and 2nd layer has two-fold advantage: i) it helps to carry the integrity from the previous layer to the following layer. This means, the network can learn new features in the 2nd layer on top of the features it learned in the 1st layer, ii) the storage requirement is highly optimized. We do not need any new Gabor kernel for the replacement of 36 weight kernels of the 2nd layer. Using different Gabor kernels in the two layers gives slightly lesser accuracy with a slightly higher storage requirement. Fig. 6.3 gives a comparative

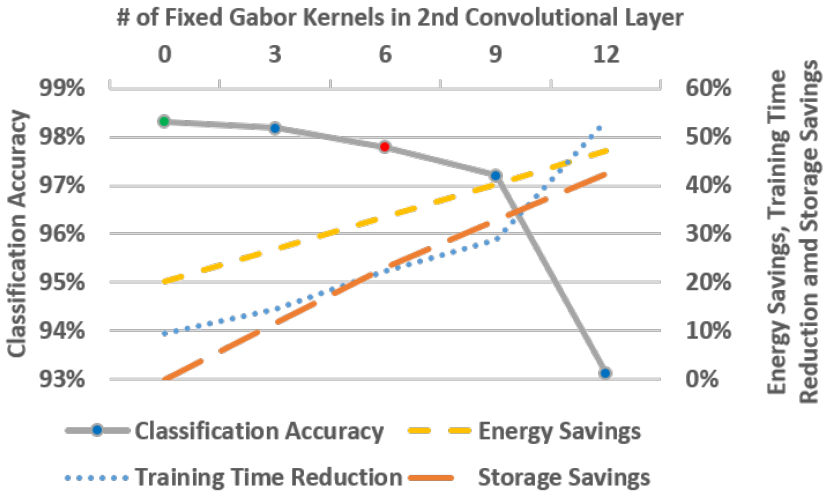
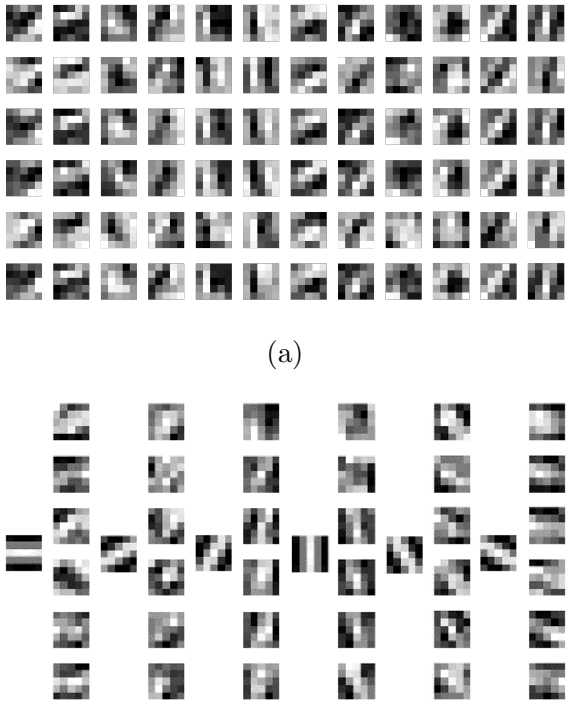


Fig. 6.2.: Change in classification accuracy, energy savings, training time reduction and storage requirement with different configurations of fixed Gabor kernels in the 2nd convolutional layer.

picture of the regular trainable kernels of the conventional CNN and half fixed/half trainable kernels of the proposed balanced system, in the 2nd layer after 100 epochs of training. The results of the Half Fixed/Half Trainable configuration are shown in Row 4 of Table 6.1. It is evident that the accuracy loss is tolerable (1.14%). Also, we see a significant reduction in energy, training time and storage requirements. From this analysis, we observe that Half Fixed/Half Trainable configuration is optimal for the network of this structure. However, other networks with different structures will have different optimal configurations. Various combination of fixed-trainable Gabor kernels can be used as a tuning knob in deeper networks to meet the energy-quality requirements.



(a)

(b)

Fig. 6.3.: (a) Trained kernels in 2nd convolutional layer and (b) Kernels of the proposed half fixed/half trainable configuration in 2nd convolutional layer.

6.3 Realization

This section describes the overall simulation framework. We used modified versions of open source MATLAB codes [45, 46] to implement multilayer CNNs for our experiments. We trained the CNNs using the corresponding training datasets mentioned in Table 6.2, to get the accuracy and training time information, which were used as a baseline for comparison. Then we introduced Gabor filters as weight kernels, and modifications in the training algorithm were made, so that the advantage of Gabor kernels can be realized. We used these newly formed CNNs for training with the corresponding datasets and collected the accuracy and training time information. We developed an energy computation model based on the number of MAC operations in the training algorithm. We implemented multiplier and adder units at the Register-

Transfer Level (RTL) in Verilog and mapped them to the IBM 45nm technology in 1 GHz clock frequency using Synopsys Design Compiler. The power and delay numbers from the Design Compiler were fed to the energy computation model to get energy consumption statistics. We also computed storage requirements and memory access energy for the overall network based on input size, number of convolutional layers, number of kernels in each layer, size of fully connected layer, number of neurons in the fully connected layer and number of output neurons. Details of the benchmarks used in our experiments are listed in Table 6.2:

Table 6.2.: Benchmarks

Application		**Dataset**	**#Training Samples**	**#Testing Samples**	**Input Size**
1.	Face Detection	Face-Nonface	600	200	48×48
2.	Digit Recognition	MNIST	60000	10000	28×28
3.	Tilburg Character Set Recog.	TICH	30000	10000	28×28
4.	Object Recognition	CIFAR10	50000	10000	32×32

CNN architectures used for the experiments are listed in Table 6.3:

Table 6.3.: CNN Architectures

Dataset	**Architecture**	**Network**
Face-Nonface	LeNet [8]	[784 (5×5)6c 2s (5×5)12c 2s 2o]
MNIST		[784 (5×5)6c 2s (5×5)12c 2s 10o]
TICH		[784 (5×5)10c 2s (5×5)20c 2s 36o]
CIFAR10 [69]	NIN [51]	[10243 (5×5)192c 160fc 96fc (3×3)mp (5×5)192c 192fc 192fc (3×3)mp (3×3)192c 192fc 10o]

*c: convolutional layer kernels, s: sub-samplingkernel,

fc: fully connected layer neurons, mp: max pooling layer kernel, o:output neurons.

6.4 Results

In this section, we present the benefits of our proposed design (with half-half balanced configuration for FaceDet., MNIST and TiCH). The results of the conventional CNN implementation, in which all the convolutional kernels are trainable, are considered as baseline for all comparisons. All the comparisons are done under iso-epoch condition.

6.4.1 Accuracy Comparison

Fig. 6.4 shows the classification accuracy obtained after 100 epochs, using conventional CNN and proposed Gabor kernel based CNN for various applications.

An interesting thing to note is that for FaceDet, the accuracy of the proposed CNN is better than the baseline. This can be attributed to the fact that FaceDet is a simpler detection task where we need to detect faces from a collection of face and non-face images in the dataset. Gabor filters being edge detectors, further simplify the face detection problem [61]. Thus, we observe better accuracy on FaceDet. In contrast, other benchmarks are multi-object classification problems where we need to

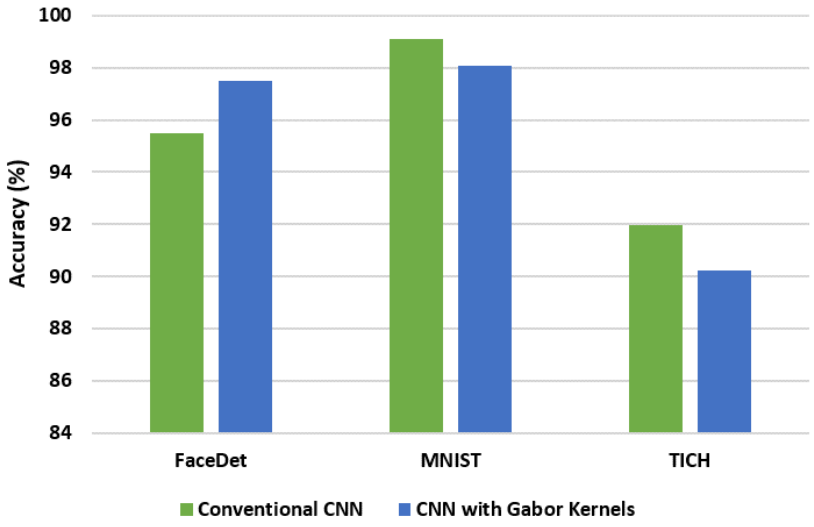


Fig. 6.4.: Comparison of accuracy between conventional CNN and Gabor kernel based CNN for different applications.

predict the class label from a collection of objects. Among the remaining benchmarks, MNIST and TiCH are character recognition datasets. The accuracy baselines for the datasets FaceDet., MNIST and TiCH are 95.5%, 99.09% and 91.98% respectively. The accuracies reported in this work are obtained using test datasets, which are separate from the training datasets for each of the benchmarks.

6.4.2 Energy Consumption Benefits

Fig. 6.5 shows the improvement in computational energy consumption achieved (during training), using Gabor kernel based CNN for different applications. We achieve 31-35% energy savings for training of CNNs for first three benchmarks mentioned in Table 6.2.

In Fig. 6.6, the pie chart represents a sample computational energy distribution during training a CNN, across different segments. The CNN used for the analysis contains two convolutional layers and two fully connected layers, and was trained on MNIST dataset. It can be seen that computation of error, loss function [45], and back

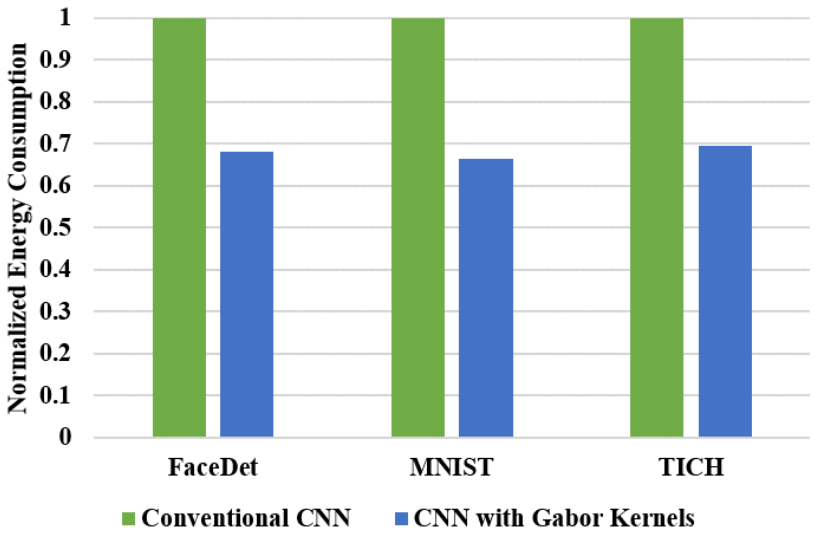


Fig. 6.5.: Comparison of energy consumption during training, between conventional CNN and Gabor kernel based CNN for different applications.

propagation of errors consume small fraction ($< 1\%$) of the total energy. A sizable portion of the energy consumption is captured by the forward propagation, gradient computation and weight update at the convolutional and fully connected layers. During back-propagation, for the 1^{st} convolutional layer this energy consumption is 20% of the total energy consumption, while for the 2^{nd} convolutional layer it is 27%. Considering that the 1^{st} convolutional layer contains only fixed Gabor kernels, the entire 20% energy consumption, required for 1^{st} convolutional layer during back-propagation (as mentioned earlier), can be saved. Also in the 2^{nd} convolutional layer, half of the kernels is trainable. That means for the 2^{nd} convolutional layer, we need 13.5% of the training energy compared to the 27% energy requirement of a conventional CNN implementation. Therefore, we can save up to 33.5% of the training energy in this particular example. This is due to the fact that we do not have to perform gradient computation or weight update for the fixed Gabor weight kernels. The amount of energy benefit is dependent on the network structure. If the fully connected layer is much larger than the convolutional layers with many hidden layers, then the energy

savings will be less. Again, if we have more convolutional layers with fixed Gabor kernels, we will get larger energy savings. In case of a deep CNN with more than two convolutional layers, the selection of the number of fixed kernels in different layers will depend on several key factors such as network structure and size, complexity of the dataset, quality requirements, among others.

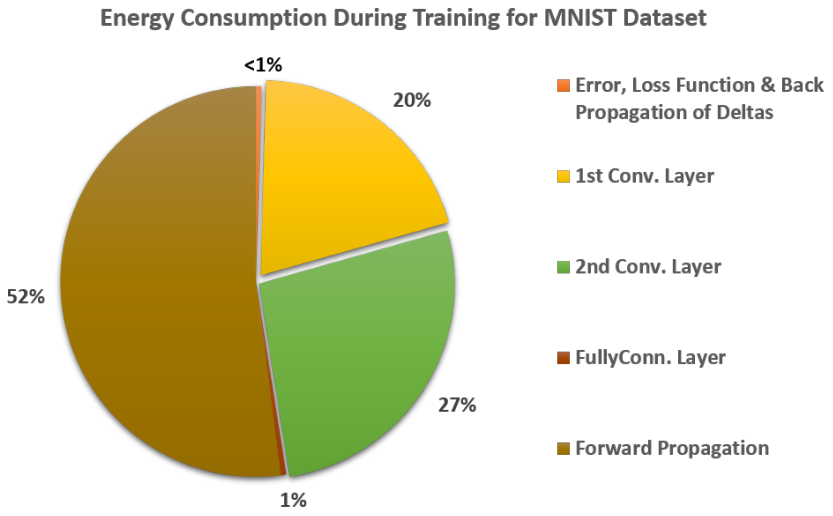


Fig. 6.6.: Energy consumption of different segments during training of a CNN with MNIST dataset.

6.4.3 Storage Requirement Reduction

Fig. 6.7 shows the storage requirement reduction obtained using proposed scheme for different applications. We achieve 15-30% reduction of storage requirement across the various benchmarks (Table 6.2). A large part of the training energy is spent on the memory read/write operations. In forward propagation, each synaptic weight requires one read operation and in back-propagation each weight update requires one write operation. We assumed on-chip storage for the feature maps. Proposed Gabor filter assisted training also provides savings in 1.2-1.3x memory access energy since

we do not need to write (update during back-propagation) the fixed kernel weights during training.

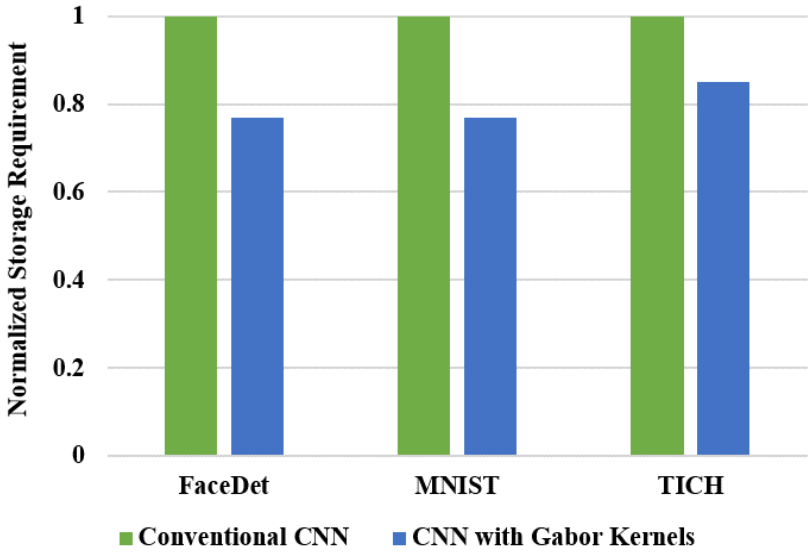


Fig. 6.7.: Comparison of storage requirements between conventional CNN and Gabor kernel based CNN for different applications.

6.4.4 Training Time Reduction

Since gradient computation and weight update is not required for the fixed Gabor kernels, we achieve significant savings in computation time with our proposed scheme. Fig. 6.8 shows the normalized training time per epoch for each application. We observe 10-29% reduction in training time per epoch across the first three benchmarks of Table 6.2. Point to be noted that the training time reduction is a by-product of our proposed method. All trainings were given same number of epochs. Number of epochs were determined ensuring that all trainings converge and reach saturation. It is observed that both conventional CNN and Gabor kernel based CNN reaches saturation in similar number of epochs. Therefore, the spare time for Gabor kernel

based CNN training cannot be used to recover accuracy loss by providing more epochs to the training.

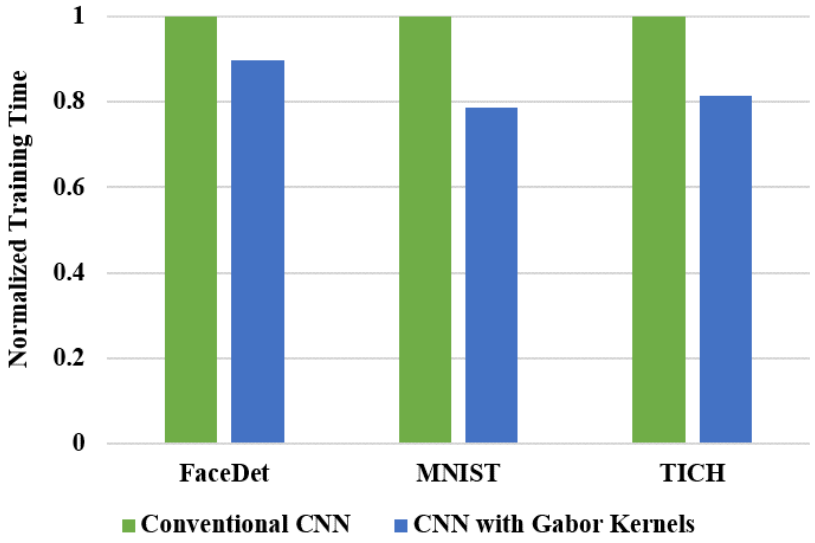


Fig. 6.8.: Comparison of training time requirements between conventional CNN and Gabor kernel based CNN for different applications.

6.4.5 Partial Training of Gabor Kernels for Accuracy Improvement

Gabor kernel based CNN provides energy savings at a cost of some accuracy degradation. For complex recognition applications, the accuracy degradation is higher. Such problems can be mitigated by employing partial training of the Gabor kernels. In the partial training, we start the CNN training with the Gabor kernels in respective positions, and then allow them to learn for first few iterations (say 20% of total number of training cycles). Since Gabor kernels provide a good starting point for the training, the kernels will learn quickly and reach near saturation within a small percentage of training cycles. Then we stop updating the partially learned Gabor kernels for the remainder of the training cycles. Point to be noted here that partial training introduces overhead, slightly increasing energy consumption, average training time, memory access energy, and storage requirement compared to the Fixed Gabor ker-

nel based CNN implementation. Nevertheless, partial training provides substantial energy saving over conventional CNN implementation, and can be a used as another tuning knob for the deep CNNs to trade-off energy-quality.

6.4.6 Applicability in Complex CNNs

Employing fixed Gabor kernels is also beneficial for larger and more complex CNNs containing more than two convolutional layers. To corroborate that, we implemented a deep CNN: [1024x3 (5×5)192c 160fc (3×3)mp (5×5)192c 192fc 192fc (3×3)mp (3×3)192c 192fc 10o] (c: convolutional layer kernels, fc: fully connected layer neurons, mp: max pooling layer kernel) for CIFAR10 dataset using MatConvNet. The network contained three MlpConv [51] blocks, each of which contained one convolutional layer, two fully connected layers and one max-pooling layer. Using 128 ($\frac{2}{3}^{\text{rd}}$ of the total 192) fixed Gabor filters as convolutional kernels in each of the convolutional layers, we achieved 29% computational energy savings and 55.7% storage reduction, while losing 3.5% classification accuracy compared to conventional CNN implementation. Though the energy savings is impressive, nevertheless the accuracy degradation is higher than desired. Therefore, we employed partial training to salvage fraction of the dropped accuracy. The Gabor kernels of the 1^{st} MlpConv block were kept fixed. The Gabor kernels of the 2^{nd} and 3^{rd} MlpConv block were partially trained for (20-30%) of total number of training cycles. This led to accuracy improvement of 1.5%, while the overhead of partial training was not substantial. The results of such training are listed in Table 6.4. Since we are training the 2^{nd} and 3^{rd} MlpConv block, all new kernel weights need to be stored. However, 2^{nd} and 3^{rd} MlpConv block were partially trained, which doesnt require heavy increase in memory access. Therefore, from 3^{rd} and 4^{th} Row of Table 6.4, we observe that storage requirement saving is only 0.5%, while memory access energy savings is 38-44%. So, the proposed scheme is scalable for deeper networks and can be employed in variety of classification tasks. The energy-

accuracy trade-off heavily depends on network structure, task complexity, sensitivity of convolutional layers, number of fixed kernels in different layers, among others.

Table 6.4.: Comparison between different Training Configurations

Configuration		**% of total Training Cycles**	**Accuracy Loss**	**Comp. Energy Savings**	**Storage Req. Savings**	**Memory Access Energy Savings**
2^{nd} Block Gabor Kernels	***3^{rd} Block Gabor Kernels***					
Fixed	Fixed	0%	3.34%	29.1%	55.2%	54.97%
Fixed	Trained	20%	2.5%	28.94%	32.31%	50.55%
Trained	Trained	20%	2.41%	25.7%	0.5%	44.19%
Trained	Trained	30%	1.95%	24.04%	0.5%	38.71%

*Accuracy loss, energy savings, storage savings and memory access energy savings are computed by considering the conventional CNN results as baseline. The baseline accuracy is 88.5%.

7. INCREMENTAL LEARNING IN DEEP CONVOLUTIONAL NEURAL NETWORKS USING PARTIAL NETWORK SHARING

Deep Convolutional Neural Networks (DCNNs) have achieved remarkable success in various cognitive applications, particularly in computer vision [11]. They have shown human like performance on a variety of recognition, classification and inference tasks, albeit at a much higher energy consumption. One of the major challenges for convolutional networks is the computational complexity and the time needed to train large networks. Since training of DCNNs requires state-of-the-art accelerators like GPUs [58], large training overhead has restricted the usage of DCNNs to clouds and servers. It is common to pre-train a DCNN on a large dataset (e.g. ImageNet, which contains 1.2 million images with 1000 categories), and then use the trained network either as an initialization or a fixed feature extractor for the specific application [70]. A major downside of such DCNNs is the inability to learn new information since the learning process is static and only done once before it is exposed to practical applications. In real-world scenarios, classes and their associated labeled data are always collected in an incremental manner. To ensure applicability of DCNNs in such cases, the learning process needs to be continuous. However, retraining these large networks using both previously seen and unseen data to accommodate new data, is not feasible most of the time. The training samples for already learned classes may be proprietary, or simply too cumbersome to use in training a new task. Also, to ensure data privacy, training samples should be discarded after use. Incremental learning plays a critical role in alleviating these issues by ensuring continuity in the learning process through regular model update based only on the new available batch of data. Nevertheless, incremental learning can be computationally expensive and time consuming, if the network is large enough.

This work focuses on incremental learning on deep convolutional neural network (DCNN) for image classification task. In doing so, we attempt to address the more fundamental issue: an efficient learning system must deal with new knowledge that it is exposed to, as humans do. To achieve this goal, there are two major challenges. First, as new data becomes available, we should not start learning from scratch. Rather, we leverage what we have already learned and combine them with new knowledge in a continuous manner. Second, to accommodate new data, if there is a need to increase the capacity of our network, we will have to do it in an efficient way. We would like to clarify that incremental learning is not a replacement of regular training. In the regular case, samples for all classes are available from the beginning of training. However, in incremental learning, sample data corresponding to new tasks become available after the base network is already trained and sample data for already learned task are no longer available for retraining the network to learn all tasks (old and new) simultaneously. Our approach to incremental learning is similar to transfer learning [71] and domain adaptation methods [72]. Transfer learning utilizes knowledge acquired from one task assisting to learn another. Domain adaptation transfers the knowledge acquired for a task from a dataset to another (related) dataset. These paradigms are very popular in computer vision. Though incremental learning is similar in spirit to transfer, multi-task, and lifelong learning; so far, no work has provided a perfect solution to the problem of continuously adding new tasks based on adapting shared parameters without access to training data for previously learned tasks.

There have been several prior works on incremental learning of neural networks. Many of them focus on learning new tasks from fewer samples [73,74] utilizing transfer learning techniques. To avoid learning new categories from scratch, Fei-Fei et al. [73] proposed a Bayesian transfer learning method using very few training samples. By introducing attribute-based classification the authors [74] achieved zero-shot learning (learning a new class from zero labeled samples). These works rely on shallow models instead of DCNN, and the category size is small in comparison. The challenge of

applying incremental learning (transfer learning as well) on DCNN lies in the fact that it consists of both feature extractor and classifier in one architecture. Polikar et al. [75] utilized ensemble of classifiers by generating multiple hypotheses using training data sampled according to carefully tailored distributions. The outputs of the resulting classifiers are combined using a weighted majority voting procedure. This method can handle an increasing number of classes, but needs training data for all classes to occur repeatedly. Inspired form [75], Medera and Babinec [76] utilized ensemble of modified convolutional neural networks as classifiers by generating multiple hypotheses. The existing classifiers are improved in [75, 76] by combining new hypothesis generated from newly available examples without compromising classification performance on old data. The new data in [75, 76] may or may not contain new classes. Another method by Royer and Lampert [77] can adapt classifiers to a time-varying data stream. However, the method is unable to handle new classes. Pentina et al. [78] have shown that learning multiple tasks sequentially can improve classification accuracy. Unfortunately, for choosing the sequence, the data for all tasks must be available to begin with. Xiao et al. [79] proposed a training algorithm that grows a network not only incrementally but also hierarchically. In this tree-structured model, classes are grouped according to similarities, and self-organized into different levels of the hierarchy. All new networks are cloned from existing ones and therefore inherit learned features. These new networks are fully retrained and connected to base network. The problem with this method is the increase of hierarchical levels as new set of classes are added over time. Another hierarchical approach was proposed in [80] where the network grows in a tree-like manner to accommodate the new classes. However, in this approach, the root node of the tree structure is retrained with all training samples (old and new classes) during growing the network.

Li and Hoiem [81] proposed 'Learning without Forgetting' (LwF) to incrementally train a single network to learn multiple tasks. Using only examples for the new task, the authors optimize both for high accuracy for the new task and for preservation of responses on the existing tasks from the original network. Though only the new

examples were used for training, the whole network must be retrained every time a new task needs to be learned. Recently, Rebuffi et al. [82] addressed some of the drawbacks in [81] with their decoupled classifier and representation learning approach. However, they rely on a subset of the original training data to preserve the performance on the old classes. Shmelkov et al. [83] proposed a solution by forming a loss function to balance the interplay between predictions on the new classes and a new distillation loss which minimizes the discrepancy between responses for old classes from the original and the updated networks. This method can be performed multiple times, for a new set of classes in each step. However, every time it incurs a moderate drop in performance compared to the baseline network trained on the ensemble of data. Also, the whole process has substantial overhead in terms of compute energy and memory.

Another way to accommodate new classes is growing the capacity of the network with new layers [84], selectively applying strong per-parameter regularization [85]. The drawbacks to these methods are the rapid increase in the number of new parameters to be learned [84], and they are more suited to reinforcement learning [85]. Aljundi et al. [86] proposed a gating approach to select the model that can provide the best performance for the current task. It introduces a set of gating auto-encoders that learn a representation for the task at hand, and, at test time, automatically forward the test sample to the relevant expert. This method performs very well on image classification and video prediction problems. However, the training of autoencoders for each task requires significant effort. Incremental learning is also explored in Spiking Neural Networks (SNN) domain. An unsupervised learning mechanism is proposed by Panda et al. [87] for improved recognition with SNNs for on-line learning in a dynamic environment. This mechanism helps in gradual forgetting of insignificant data while retaining significant, yet old, information thus trying to address catastrophic forgetting.

In the context of incremental learning, most work has focused on how to exploit knowledge from previous tasks and transfer it to a new task. Little attention has

gone to the related and equally important problem of hardware and energy requirements for model update. Our work differs in goal, as we want to grow a DCNN with reduced effort to accommodate new tasks (set of classes) by network sharing, without forgetting the old tasks (set of classes). The key idea of this work is the unique 'clone-and-branch' technique which allows the network to learn new tasks one after another without any performance loss in old tasks. Cloning layers provides a good starting point for learning a new task compared to randomly initialized weights. The kernels learn quickly, and training converges faster. It allows us to employ fine-tuning in the new branch, saving training energy and time compared to training from scratch. On the other hand, branching allows the network to remember task specific weight parameters, hence the network does not forget old tasks (in task specific scenario) no matter how many new tasks it has learned. The novelty of this work lies in the fact that we developed an empirical mechanism to identify how much of the network can be shared as new tasks are learned. We also quantified the energy consumption, training time and memory storage savings associated with models trained with different amounts of sharing to emphasize the importance of network sharing from hardware point of view. Our proposed method is unique since it does not require any algorithmic changes and can be implemented in any existing hardware if additional memory is available for the supplementary parameters needed to learn new classes. There is no overhead of storing any data sample or statistical information of the learned classes. It also allows on-chip model update using a programmable instruction cache. Many of the state-of-the-art DNN accelerators support this feature. However, FPGAs are the kind of hardware architecture that is best suited for the proposed method. It offers highly flexible micro-architecture with reusable functional modules and additional memory blocks in order to account for dynamic changes.

In summary, the key contributions of our work are as follows:

- We propose sharing of convolutional layers to reduce computational complexity while training a network to accommodate new tasks (sets of classes) without forgetting old tasks (sets of classes).

- We developed a methodology to identify optimal sharing of convolutional layers in order to get the best trade-off between accuracy and other parameters of interest, especially energy consumption, training time and memory access.
- We developed a cost estimation model for quantifying energy consumption of the network during training, based on the Multiplication and Accumulation (MAC) operations and number of memory access in the training algorithm.
- We substantiate the scalability and robustness of the proposed methodology by applying the proposed method to different network structures trained for different benchmark datasets.

We show that our proposed methodology leads to energy efficiency, reduction in storage requirements, memory access and training time, while maintaining classification accuracy without accessing training samples of old tasks.

7.1 Incremental Learning

A crude definition of incremental learning is that it is a continuous learning process as batches of labeled data of new classes are gradually made available. In literature, the term "incremental learning" is also referred to incremental network growing and pruning or on-line learning. Moreover, various other terms, such as lifelong learning, constructive learning and evolutionary learning have also been used to denote learning new information. Development of a pure incremental learning model is important in mimicking real, biological brains. Owing to superiority of biological brain, humans and other animals can learn new events without forgetting old events. However, exact sequential learning does not work flawlessly in artificial neural networks. The reasons can be the use of a fixed architecture and/or a training algorithm based on minimizing an objective function which results in *"catastrophic interference"*. It is due to the fact that the minima of the objective function for one example set may be different from the minima for subsequent example sets. Hence each successive training set causes

the network to partially or completely forget previous training sets. This problem is called the *"stability-plasticity dilemma"* [88]. To address these issues, we define an incremental learning algorithm that meets the following criteria:

i. It should be able to grow the network and accommodate new tasks (sets of classes) that are introduced with new examples.

ii. Training for new tasks (sets of classes) should have minimal overhead.

iii. It should not require access to the previously seen examples used to train the existing classifier.

iv. It should preserve previously acquired knowledge, *i.e.* it should not suffer from catastrophic forgetting.

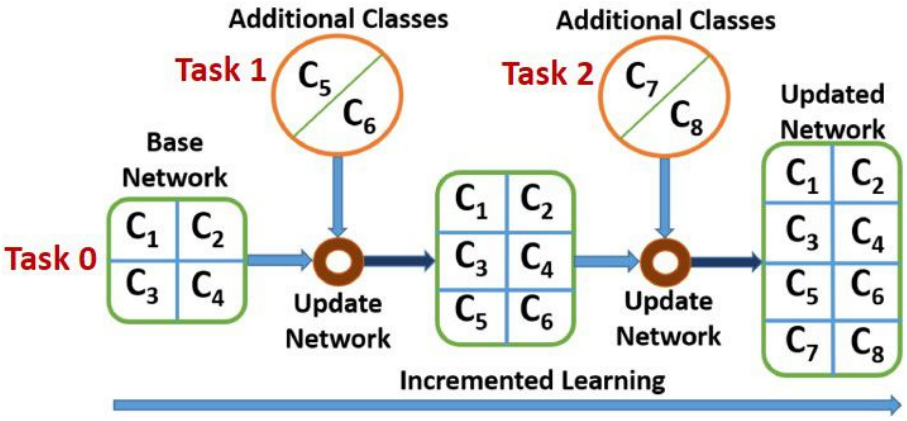


Fig. 7.1.: Incremental learning model: the network needs to grow its capacity with arrival of data of new tasks (sets of classes).

In this work, we developed an efficient training methodology that can cover aforementioned criteria. Let us comprehend the concept with a simple example. Assume that a base network is trained with four classes of task 0 ($C_1 - C_4$), and all training data of those four classes are discarded after training. Next, sample data for task 1 with two classes ((C_5, C_6)) arrive and the network needs to accommodate them while keeping knowledge of the initial four classes. Hence, the network capacity has to be increased and the network has to be retrained with only the new data of task

1 (of C_5 and C_6) in an efficient way so that the updated network can classify both task's classes ($C_1 - C_6$). If the tasks are classified separately, then it is a task specific classification. On the other hand, when they are classified together, then it is called combined classification. We will primarily focus on the task specific scenario while also considering the combined classification. Fig. 7.1 shows the overview of the incremental learning model we use.

7.1.1 Advantages

There are several major benefits of incremental learning.

Enable training in low power devices

Training a deep network from scratch requires enormous amount of time and energy which is not affordable for low power devices (embeded systems, mobile devices, IoTs etc.). Therefore, a deep network is trained off-chip and deployed in the edge devices. When data for new task are available, it can not be used for learning in the device because of two reasons; i) the device does not have access to sample data for already learned tasks, ii) the device does not posses the capability to retrain the whole network. However, the new tasks can be learned incrementally by reusing knowledge from existing network without requiring data samples of old tasks. This enables the low power devices to update the existing network by incrementally retraining it within their power budget and hardware limitations.

Speed up model update

If knowledge from existing network can be reused while learning new tasks (with new data samples only) without forgetting old tasks, then the updating process of an existing network will be very fast.

Ensure data privacy

Incremental learning do not require access to old training data. Therefore, all training samples can be discarded after each training session, which will disallow misuse of private data.

Reduce storage requirements

Deep networks require humongous amount of data to train. Since training data samples are not required to be stored for incremental learning, the storage requirement for updating a network is greatly reduced.

The following section will describe the design approach of the proposed scheme.

7.2 Design Approach

The superiority of DCNNs comes from the fact that it contains both feature extractor and classifier in the same network with many layers. '*Sharing*' convolutional layers as fixed feature extractors is the base of our proposed training methodology. '*Sharing*' means reusing already learned network parameters/layers to learn new set of classes. Note that in all cases, while learning new classes, only newly available data is used. Also, we assume that new classes will have similar features as the old classes. Therefore, we separate a single dataset into several sets so that they can be used as old and new data while updating the network. All accuracies reported in this work are test accuracies (training samples and test samples are mutually exclusive).

This section outlines the key ideas behind the proposed methodology.

7.2.1 Increasing Convolutional Kernels in the Last Layer

To accommodate more classes, the network must increase its capacity. The simplest way to do that is widening the final *softmax* layer to output the extra probabilities for the new classes. One obvious drawback of this approach is that the increment

of learning capacity is small [79]. For example, let us consider a small CNN with two convolutional layers each containing 4 (5×5) convolutional kernels and a fully connected layer to connect the feature vector output with the output neurons. If the initial number of classes is 10, and 5 new classes are to be accommodated, then the increase in trainable parameters is only 17.8% compared to 50% increase in the number of classes. Since we do not want to forget the already learned classes, we can only train the small percentage of trainable parameters with the new examples. However, this does not result in a good inference accuracy for the new classes. For a large network with many convolutional layers, the increment of learning capacity reduces further and goes as low as less than 1%.

Therefore, it is prudent to widen the network by having more feature maps in the convolutional layers. To investigate this idea, we trained the above-mentioned network denoted by [784 (5×5)4c 2s (5×5)4c 2s 10o] (CNN containing 784 input neurons, 2 convolutional layers (4c) each followed by a sub-sampling layer (2s), and finally a fully connected layer with 10 output neurons (10o)), for 10 classes (digits 0-9 from TiCH dataset [68]). Then we added rest of the 26 classes (alphabets) to the existing network in five installments. Each time we retrain the network for new classes, we add two feature maps in the last convolutional layer. For example, when we add first 5 classes (A-E) with the existing 10 classes, we retrain the network [784 (5×5)4c 2s (5×5)**6**c 2s **5o**]. We only increment the concluding convolutional layer since it has been shown by Yosinski et al. [66] that initial layers in a DCNN is more generic while last layers are more specific. Therefore, we only focus on incrementing and retraining the last few layers. Note that we added specifically 2 feature maps in the last convolutional layer (for each addition of 5-6 classes) in order to increase the model capacity while maintaining the existing class/filter ratio ($\sim$2.5 classes/filter) and prevent over-fitting. The new parameters are initialized using random numbers which have distribution similar to the learned weights. Cloning weights from learned filters provide similar results.

In the retraining process, only the 8 kernels corresponding to the 2 new feature maps and connections to the 5 new output neurons are trained with the new examples. Rest of the parameters are shared with the base network of 10 classes and as they are frozen, we will not forget the previously learned 10 classes. The new network becomes base network for the next 5 classes (F-J) to be added. That means, for any new set of classes, the network will be [784 (5×5)4c 2s (5×5)**8**c 2s **5o**], where only the 8 kernels corresponding to the 2 new feature maps and connections to the 5 new output neurons will be trained with the new examples. The accuracy achieved by this approach is given in the Table 7.1.

Table 7.1.: Accuracy results for approach 1

Classes	Network	Incremental Learning Accuracy (%)	
		With partial network sharing	Without partial network sharing
0-9 (base)	[784 (5×5)4c 2s (5×5)4c 2s 10o]	–	96.68
A-E	[784 (5×5)4c 2s (5×5)**6**c 2s **5**o]	98.50	98.82
F-J	[784 (5×5)4c 2s (5×5)**8**c 2s **5**o]	98.95	99.90
K-O	[784 (5×5)4c 2s (5×5)**10**c 2s **5**o]	98.03	98.14
P-T	[784 (5×5)4c 2s (5×5)**12**c 2s **5**o]	98.17	98.41
U-Z	[784 (5×5)4c 2s (5×5)**14**c 2s **5**o]	96.57	96.76

We can observe from Table 7.1 that the accuracy degradation due to *'partial network sharing'* is negligible compared to the network *'without partial sharing'*. Note that *'without partial network sharing'* is the case when new classes are learned using all trainable parameters, none of which are shared with the already learned network parameters. In the case of training *'without partial network sharing'*, the new layers are initialized using the model with data A (old), and then fine-tuned with data B (new) without freezing any parameters.

However, such an approach has scalability issues. If we keep on adding more classes and continue increasing feature maps in the last convolutional layer, the network will become inflated towards the end. And hence, there can be overfitting and convergence issues while retraining for the new set of classes. We take care of this problem by retraining the final convolutional layer completely, the details of which are described in the following section.

7.2.2 Adding Branch to Existing Network

The approach presented in section 7.2.1 is a straight forward one and shown in earlier related works. However, the limitations of such approach has motivated our search for a robust scalable method.

To learn new set of classes, we *clone* and retrain the final convolutional layer and subsequent layers. To *clone* a layer or layers, we create a new layer or layers with the exact same number of neurons as in the original layer/layers and initialize the new layer/layers with the same weight values so that both original and the cloned layer/layers have exactly same synaptic connections. A new network is formed every time we add new set of classes, which shares the initial convolutional layers with the base network, and has a separate final convolutional layer and layers following it to the output. The cloned and retrained layers of the new network thus become a branch of the existing network. The advantage of cloning the final layers is that we do not have to worry about the initialization of the new trainable parameters. Otherwise, new kernels initialized with too big or too small a random value will either ruin the existing model or make training tediously long. Another advantage of cloning is that it maximizes the transfer of learned features.

To investigate this approach, we implemented a deep CNN: [1024×3 (5×5)128c (1×1)100c (1×1)64c (3×3)mp (5×5)128c (1×1)128c (1×1)128c (3×3)mp (3×3)128c (1×1)128c (1×1)10o] (c: convolutional layer kernels, mp: max pooling layer kernel, o: output layer) for CIFAR-10 [69] dataset using MatConvNet [46]. The network

contained three MlpConv [51] blocks, each of which contained one convolutional layer consisting of 5×5 or 3×3 kernels, two convolutional layers consisting of 1×1 kernels and one max-pooling layer. The (1×1) convolutional layers can be considered as fully-connected layers. Hence, 'final convolutional layer' implies the convolutional layer in the last MlpConv block that contains 3×3 kernels.

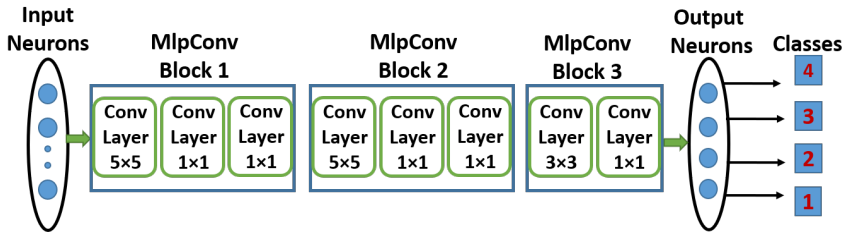
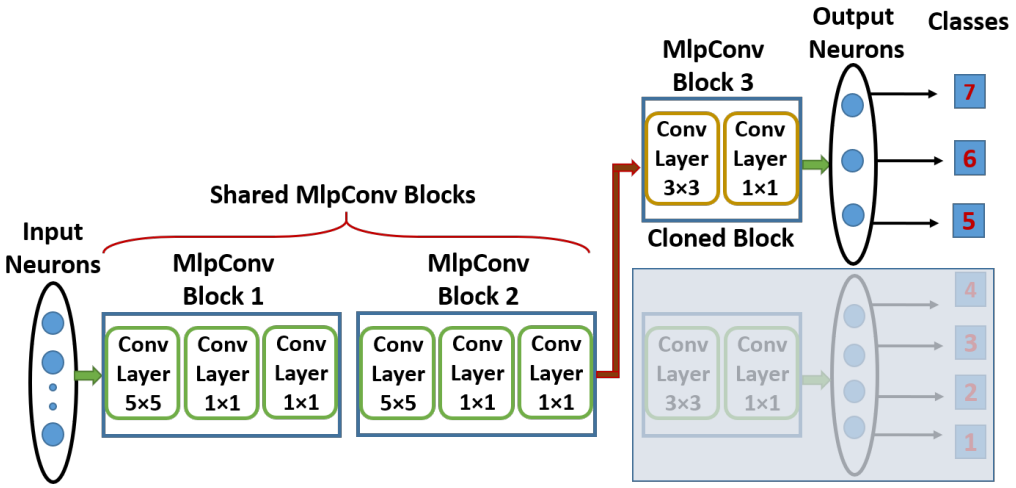
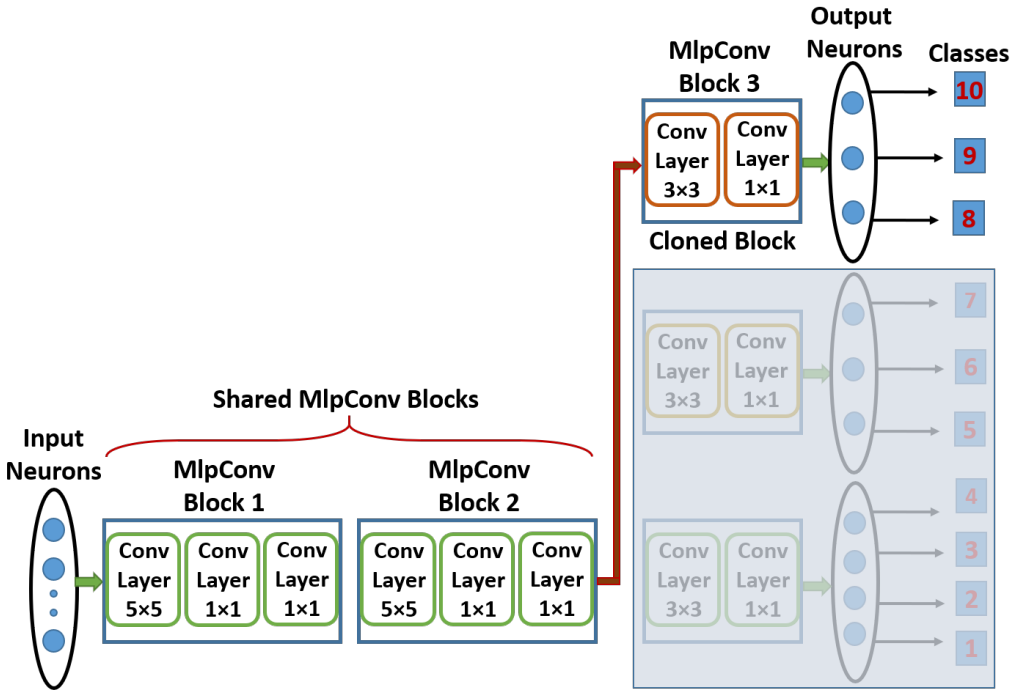


Fig. 7.2.: Network structure for investigating incremental learning by retraining the final convolutional layer.

First, we separated the 10 classes of CIFAR-10 dataset to three sets of 4, 3 and 3 classes. The classes were chosen randomly. We trained the base network using the set of 4 classes. Fig 7.2 shows the basic structure of the network. The last layer of the final MlpConv block is shown separately in the figure to specify it as the output layer. The max-pooling layers and final average pooling layer is not shown for simplicity. After training this base network, we added rest of the two set of classes to the existing network in two installments. Each time we retrain the network for new classes, we clone the last MlpConv [51] block (highlighted in Table 7.2) and retrain it using new examples for the new set of classes. During this retraining, the initial two MlpConv blocks are shared from the base network which work as fixed feature extractors (learning parameters are frozen) and minimize learning overhead for new set of classes (Fig. 7.3). In Fig. 7.3a, the new MlpConv block is cloned from the base network and only that part is retrained with the new data samples for the new classes, while the last MlpConv block of the base network remains disconnected. Similarly, another branch is trained for the next set of new classes as shown in Fig. 7.3b. After



(a)



(b)

Fig. 7.3.: Incremental training for accommodating (a) first and (b) second set of new classes in the base network. The green blocks imply layers with frozen parameters. The semi-transparent rectangle implies that the part is disconnected from training.

retraining, the new MlpConv block is added to the existing network as a new branch. Fig. 7.4a shows the updated network after adding the two sets of new classes.

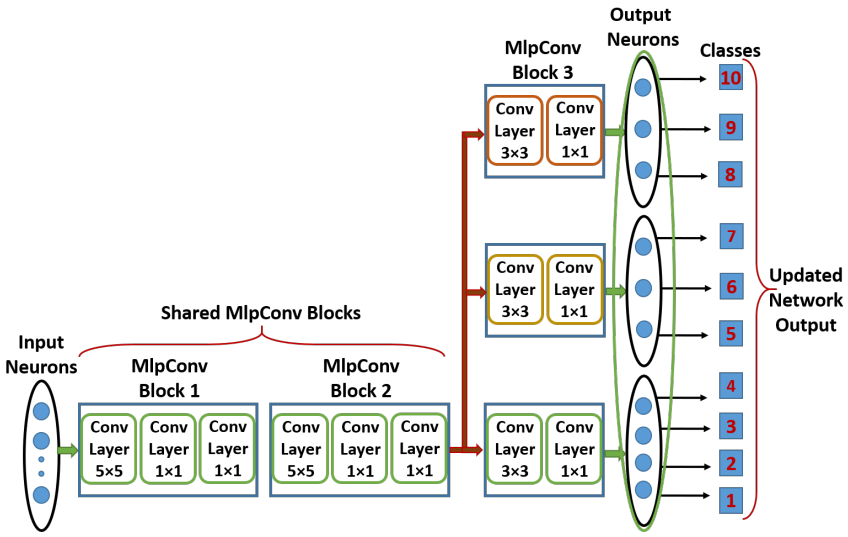
Table 7.2.: Accuracy results for approach 2

Classes	Network	Incremental Learning Accuracy (%)	
		With partial network sharing	Without partial network sharing
10 (all classes)	[1024 × 3 (5 × 5)128c (1 × 1)	–	88.90
4 (base)	100c (1 × 1)64c (3 × 3)mp	–	91.82
3	(5 × 5)128c (1 × 1)128c (1 × 1)	89.60	90.53
3	128c (3 × 3)mp **(3 × 3) 128c**	96.07	96.40
10 (updated)	**(1 × 1)128c 4/3/3/10o**]	58.72	60.49

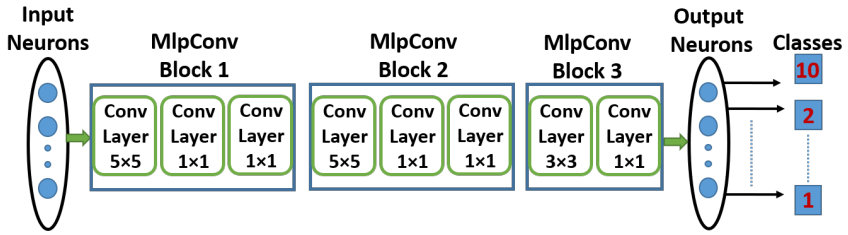
We can observe from the Table 7.2 that the accuracy degradation due to partial network sharing (Fig. 7.3) is negligible when we train for additional class sets. On the other hand accuracy for updated network (Fig. 7.4a) of 10 classes suffers ∼1.8% degradation compared to an incremental learning approach where we do not share the first two MlpConv blocks for learning the new classes. In the case of learning *w/o sharing MlpConv blocks*, each new branch is trained separately with 3 MlpConv blocks rather than 1 final block.

We would like to mention that ∼89% (row 1, column 4 in Table 7.2) classification accuracy can be achieved for CIFAR-10 dataset using slightly modified NIN [51] architecture (Fig. 7.4b), if the training is done with all training samples applied together as in regular training. This performance can be considered as the upper-bound for incremental learning on this network, for this particular dataset. However, for incremental learning, all training samples are not available together, hence it is not possible to get that high accuracy even without any network sharing.

Note that freezing a set of parameters in a pre-trained convolutional network is a standard practice for many applications involving knowledge transfer. But previous



(a)



(b)

Fig. 7.4.: (a) Updated network after incrementally learning two sets of new classes. (b) Modified NIN [51] architecture for training CIFAR-10 dataset with all training samples (regular training).

works [71] used this method to learn a different dataset using the frozen parameters as fixed feature extractors. In such case, the new network can only classify the new dataset, not previously learned dataset. On the other hand, in our proposed methodology, both old and new learned classes can be classified together as well as separately. However, one question is still unanswered: in a large DCNN with many convolutional layers, is retraining the final convolutional layer enough? To answer this question, we move to our third and final approach which will be described in the next sub-section.

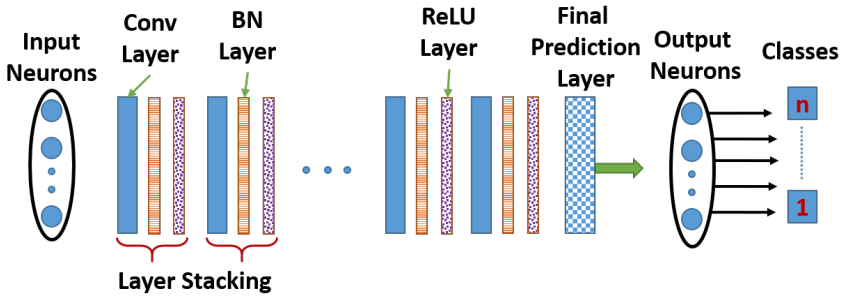
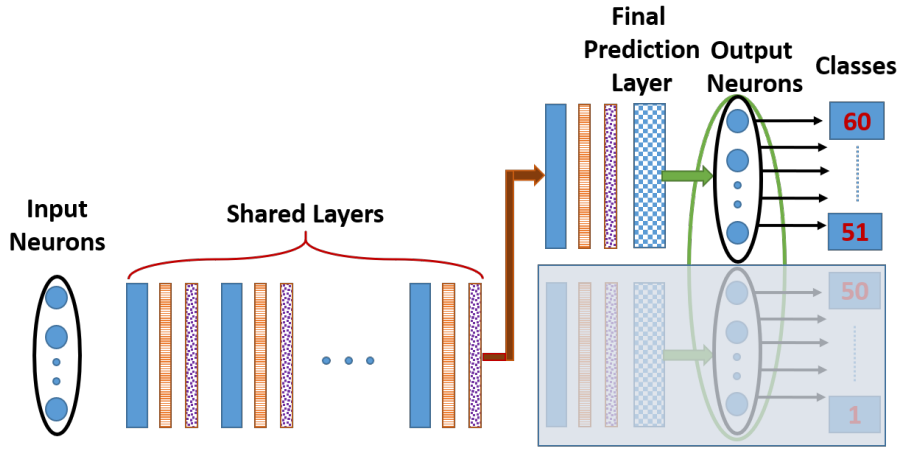


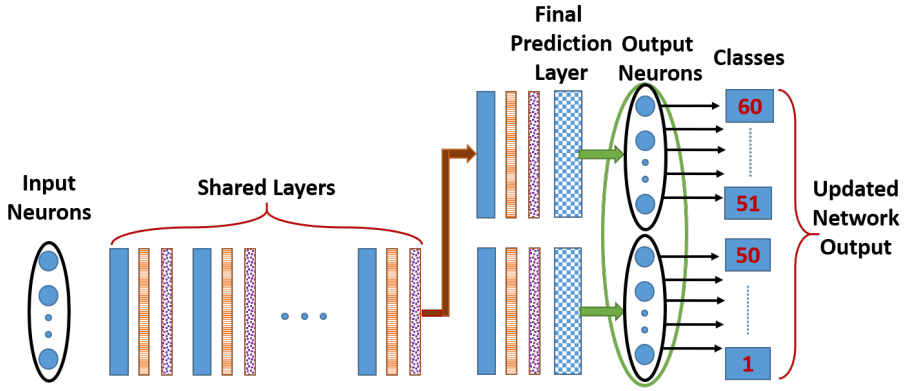
Fig. 7.5.: The ResNet [39] network structure used for implementing large scale DCNN. For simplicity, the input bypass connections of ResNet is not shown here.

7.2.3 Replacing Part of the Base Network with New Convolutional Layers

A large DCNN usually has many convolutional layers followed by a fully connected final classifier. To apply our approach in a large DCNN, we implemented ResNet [39] for a real-world object recognition application. The network structure is depicted in Fig. 7.5. CIFAR-100 [69] was used as the benchmark dataset. We trained a base network (ResNet50), with 50 classes out of the 100 in CIFAR-100. Then we added rest of the 50 classes to the existing network in three installments of 20, 20 and 10. The classes for forming the sets were chosen randomly and each set is mutually exclusive. Each time when we update the network for new tasks, we clone the last convolutional layer and following classifier layers, while sharing the initial convolutional layers from the base network, and retrain it using examples for the new set of classes only (Fig. 7.6a). After retraining the cloned branch, we add it to the existing network as a new branch as shown in Fig. 7.6b. Note that, the initial part of the base network is shared and frozen. After training the branch network for new task (additional classes), we have the updated network that can do task specific classification as well as combined classification. During task specific classification, only the task specific branch will be active, while for combined classification all branches will be active at the same time. Since during training the new branches, shared and old task specific parameters are not altered, the network will not forget already learned tasks. However, training only 1



(a)

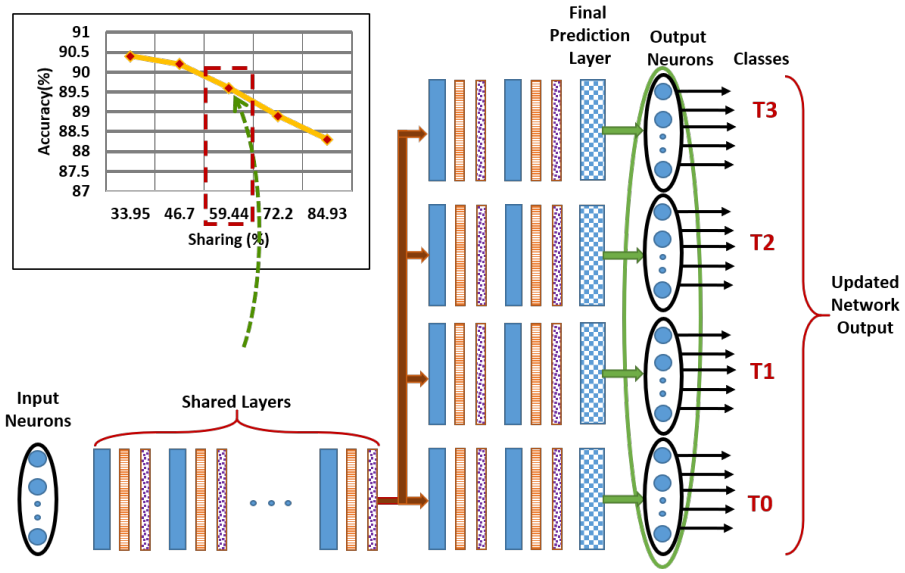


(b)

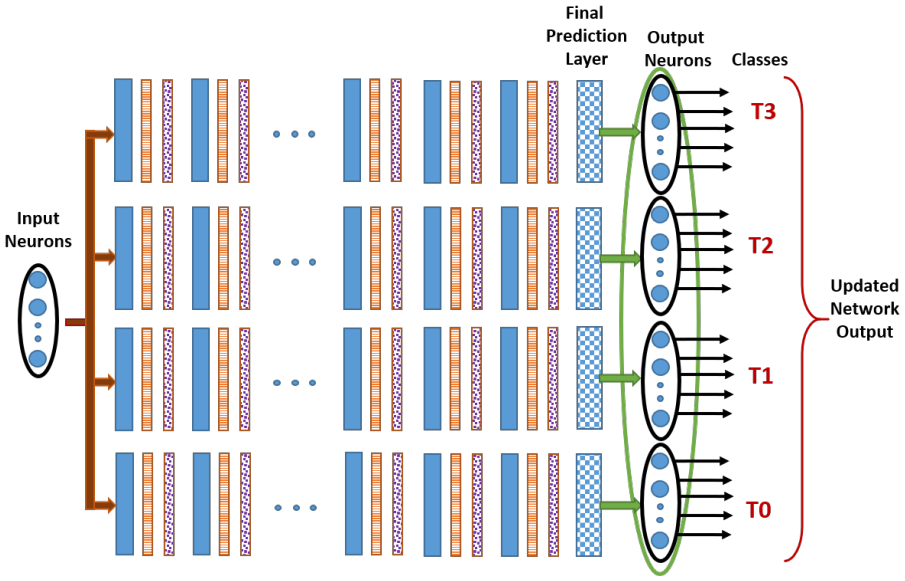
Fig. 7.6.: (a) Incremental training for accommodating new classes in the base network. The parameters of the shared layers are frozen. The semi-transparent rectangle implies that the part is disconnected from training. The new convolutional layer is cloned from the base network and only that part is retrained with the new data samples for the new classes, while the last convolutional layer of the base network remain disconnected. (b) After retraining the cloned layer, we add it to the existing network as a new branch, and form the updated network.

conv layer and classifier layers is not enough to learn a new task properly. Hence, new task performance suffers in this configuration. We compared the accuracies achieved

by this method with the accuracy of a network of same depth, trained without sharing any learning parameter, and observed that there is an 8-12% accuracy degradation for the former method. We also assessed the updated network accuracy for the all 100 classes by generating prediction probabilities from each of the separately trained networks and selecting the maximum probability. Even for the updated network, we observed about 10% accuracy degradation. To counter this accuracy degradation, we developed a training methodology that will be described in the following subsection.



(a)



(b)

Fig. 7.7.: (a) Updated network architecture for proposed training methodology. '%' Sharing is the portion of trainable parameters which are frozen and shared between the base and the new network. This quantity is decided from the *'Accuracy vs Sharing'* curve shown in the inset. (b) It is an incrementally trained network, without network sharing, used as baseline for comparison.

7.2.4 Training Methodology 1

To mitigate the accuracy loss due to sharing, we reduced network sharing and allowed more freedom for retraining the branch networks. By gradually reducing sharing we observed improvement in the inference accuracy for both branch networks and the updated network.

From Fig. 7.7a, we can observe that when we share ∼60% of the learning parameters in the convolutional layers (and corresponding batch normalization and ReLU layers), we can achieve accuracy within ∼1% of baseline. The baseline is an incrementally trained network, without network sharing (Fig. 7.7b). The accuracy results for this network configuration is listed in Table 7.3. Note that ∼73% classification accuracy (row 1, column 4 in Table 7.3) can be achieved for CIFAR-100 using ResNet50, which is the upper-bound for incremental learning on this network, if the training is done with all training samples applied together. But for incremental learning, all training samples are not available together, hence it is not possible to get that high accuracy even without any network sharing. If we share more than 60% of the network parameters, classification accuracy degrades drastically. Based on this observation we developed the incremental training methodology for maximum benefits.

Table 7.3.: Accuracy results for Training Methodology 1

Classes	Network	Incremental Learning Accuracy (%)	
		With partial network sharing	Without partial network sharing
100 (all classes)	ResNet50 [39]:	–	73.95
50 (base)	43 Convolution,	–	77.02
20	40 Batch Normalization,	85.65	85.80
20	40 ReLU, 1 average pooling,	84.05	88.00
10	1 Output Prediction layer	93.50	94.10
100 (updated)		59.51	61.00

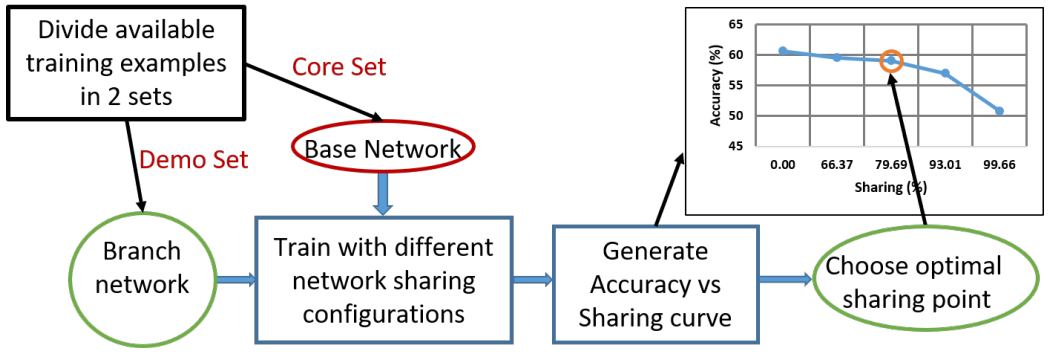


Fig. 7.8.: Overview of the DCNN incremental training methodology with partial network sharing.

We propose an incremental training methodology with optimal network sharing as depicted in Fig. 7.8. The initially available set of classes are divided in to 2 sets. The larger or Core set is used to train a base network. Then the smaller set, which we call a demo set, is used to train a cloned branch network with different sharing configurations. From the training results, an *Accuracy vs Sharing* curve is generated, from which the optimal sharing configuration for this application and this network architecture is selected. This curve shows how much of the initial layers from the base network can be shared without severe accuracy degradation on the new task. An optimal sharing configuration is selected from the curve that meets quality specifications. This optimal configuration is then used for learning any new set of classes.

There is an overhead for determining the optimal sharing configuration from the accuracy-sharing trade-off curve. This curve provides a tuning knob for trading accuracy with energy benefits. However, we do not need to explore the entire search space and we can apply heuristics based on network architecture, number of trainable parameters, dataset complexity, number of training samples etc. to find the optimal sharing configuration within few iterations of retraining the cloned network. Note that training of cloned network is fast and low cost as the shared layers are not back-

propagated. In the following paragraph, we will describe the optimal sharing point determination procedure.

To train a base network, we separate initially available classes in two sets: Core set and Demo set. Then we train the base network with the core set and a separate network with the demo set. Accuracy of this separate network will be used as reference for determining the optimal sharing configuration. Next, we create a branch network (that will share some initial layers from the base network) and train it for classes in the demo set. This branch network is a cloned version of the trained base network. The amount of the network sharing can be initially chosen based on the heuristics discussed earlier. For instance, we chose to share 50% of ResNet50 parameters for CIFAR-100 dataset. Then we train the branch network and compare its performance with the reference accuracy. If the new accuracy is close to the reference, then we increase sharing and train the branch again to compare. On the other hand, if the new accuracy is less than the reference, then we decrease sharing and train again to compare. After few iterations, we finalize the optimal sharing configuration based on the required quality specifications. The optimal sharing point is the sharing fraction, beyond which the accuracy degradation with increased sharing is higher than the quality threshold. This leads to maximal benefits with minimal quality loss. Finally, we can retrain the base network with both sets (core and demo) together to improve the base network features (in the initial layers), since both sets (core and demo) are available. This base network training and optimal sharing configuration analysis should be done off-chip assuming that there is no energy constraint. Then this base network can be deployed on energy-constrained device (edge) where new classes will be learned. The overhead of optimal sharing configuration selection by generating the accuracy-sharing curve is a onetime cost and it can be neglected since it will be done off-chip.

For inference, under the separate task scenario, it will be a two stage network. The multi-stage network will allow selective activation of a task specific branch [89] while other branches will be inactive. For the combined classification scenario, all

branches will remain active at the same time. Note in Fig. 7.7a in the top layers, there are branches for old and new set of classes. While retraining, and updating the network for new set of classes, only the branch of top layers corresponding to the new set of classes are retrained. Thus, the top layer filters keep information of their respective set of classes and the network do not suffer from catastrophic forgetting.

In this work, we do not try to grow a model with classes from datasets of different domains since the base network have learned features from data samples of a single dataset. For instance, if the base network is trained on object recognition dataset CIFAR-10, then it will be able to accommodate new classes from CIFAR-100 dataset as both of the datasets have similar type of basic features (image size, color, background etc.). However, the same base model should not be able to properly accommodate new classes from character recognition dataset (MNIST) because MNIST data has very different type of features compared to CIFAR-10 data.

7.2.5 Training Methodology 2

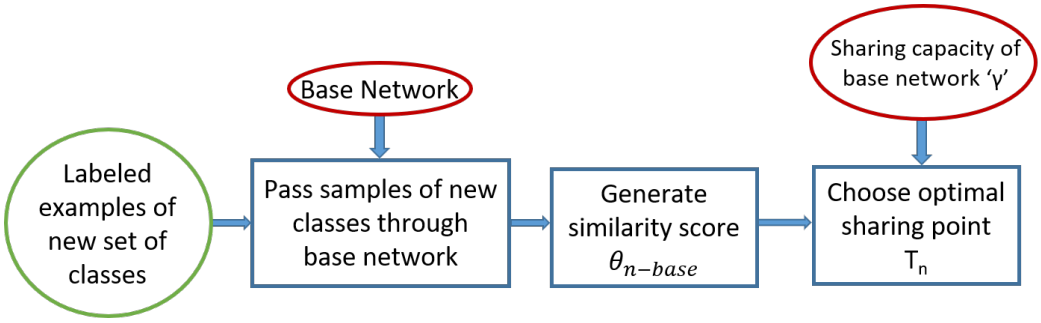


Fig. 7.9.: Incremental training methodology for task specific partial network sharing.

In training methodology 1, all branches of the updated network has equal number of task specific parameters. However, it is not necessary to have this constraint while learning a new task. We extend training methodology 1 to implement task specific sharing configuration as shown in Fig. 7.9. In this case, we forward propagate few samples of a new task through the trained base network. From classification results of

those samples, we generate a similarity score that approximately quantifies similarity between the classes of base network and the new task. To generate the similarity score we have used algorithm 3. We pass random samples of a class belonging to a new task through the trained base network. From the classification results, we count number of repeating classes as similarity points. We repeat this process for all the different classes of the new task several times and take the average number as the similarity score. This is a very simple way to measure similarity, however may not be an ideal one. We employed this method considering its simplicity, so that the overhead of measuring similarity score do not overtake the advantages of task specific sharing.

ALGORITHM 3: Similarity score generation

Input: Trained Base network: Base NN, New task data: TnData.

Output: Similarity score of new task with respect to learned task: θ_{n-base}

1. Randomly sample 5 training examples for each new class in the new task and forward propagate through the trained base network for classification.
2. Count number of repeating classes as similarity.
3. Repeat the steps 1 and 2, 3 times and average the results to get average similarity score.

Next, we estimate the sharing capacity of the base network from the optimal sharing configuration of the base network and the similarity score of the demo set that was used to generate the 'Accuracy-sharing' curve. Then we use equation 1 to estimate task specific sharing configuration. The task that have higher similarity with the base network, will be able to share more features from the base network. Note that each network architecture has different sharing capacity. Hence, it is necessary to estimate the sharing capacity of the base network using training methodology 1.

$$sharing\ for\ task\ `n',\ T_n = \gamma \times \theta_{n-base} \tag{1}$$

where, γ is the sharing capacity of the base network and θ_{n-base} is the similarity score between the new task and the base network.

$$\gamma = \frac{optimal\ sharing\ of\ base\ network}{\theta_{base}} \tag{2}$$

Plugging in the network learning capacity and similarity scores in equation 1, we generate a look-up table from which sharing configuration for new task will be determined. The updated network for CIFAR-100 shown in the Fig. 7.10 is the result of this approach.

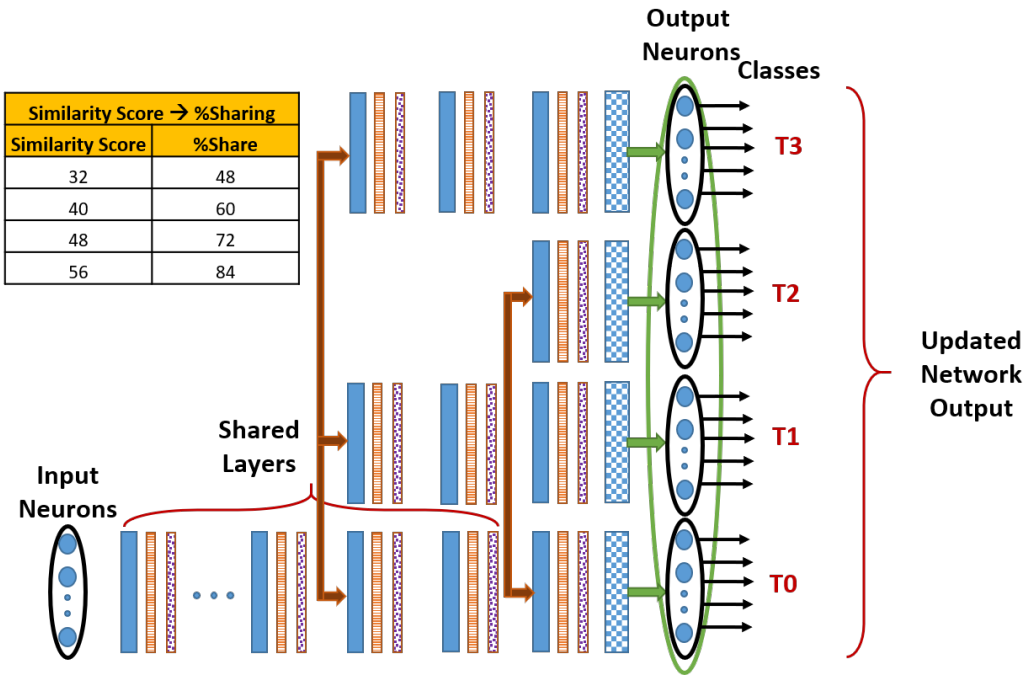


Fig. 7.10.: Updated network for task specific partial network sharing using similarity score table.

7.2.6 Training Methodology 3

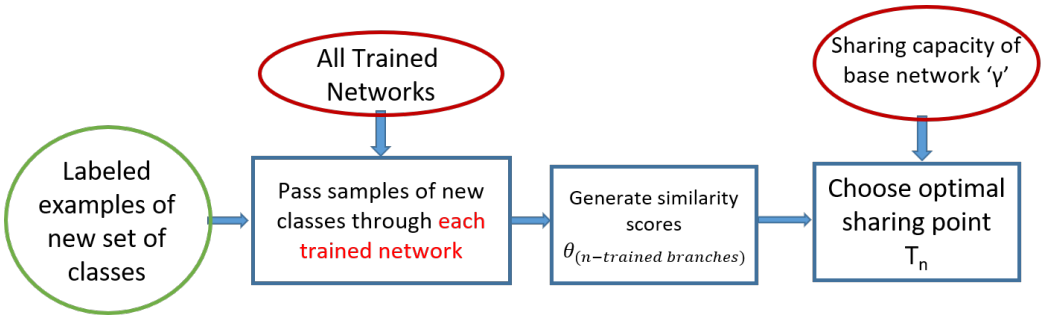


Fig. 7.11.: Incremental training methodology for fine grain optimization.

We can employ a more fine grain optimization by sharing not only from the base network, but also from the trained branches. The training methodology is depicted in Fig. 7.11. In this method, we have to generate similarity score of the new task with all previously trained task classes using equation 3. New task branch will share from the most similar branch.

$$sharing\ for\ task\ 'n',\ T_n = \gamma \times\ max\theta_{n-trained\ networks} \tag{3}$$

where, γ is the sharing capacity of the base network and $max\theta_{n-trained\ networks}$ is the maximum similarity score between new task and trained network branches.

Let us see an example (Fig. 7.12a). Assume that the base network is trained with task 0. When data for task 1 becomes available, it has the only option to share network with task 0. Then, data for task 2 arrives and it has two options, share network with task 0 or task 1. From the similarity score table, we can observe that task 2 has higher similarity with task 1 than task 0. So, branch network for task 2 shares network with task 1. For task 3, there are 3 options and it shares with task 1 since it has higher similarity score with task 1. Note that, task order plays an important role in this approach. For instance, in this specific example, if task 2 is available before task 1 or task 3, we will get a different updated network (Fig. 7.12b).

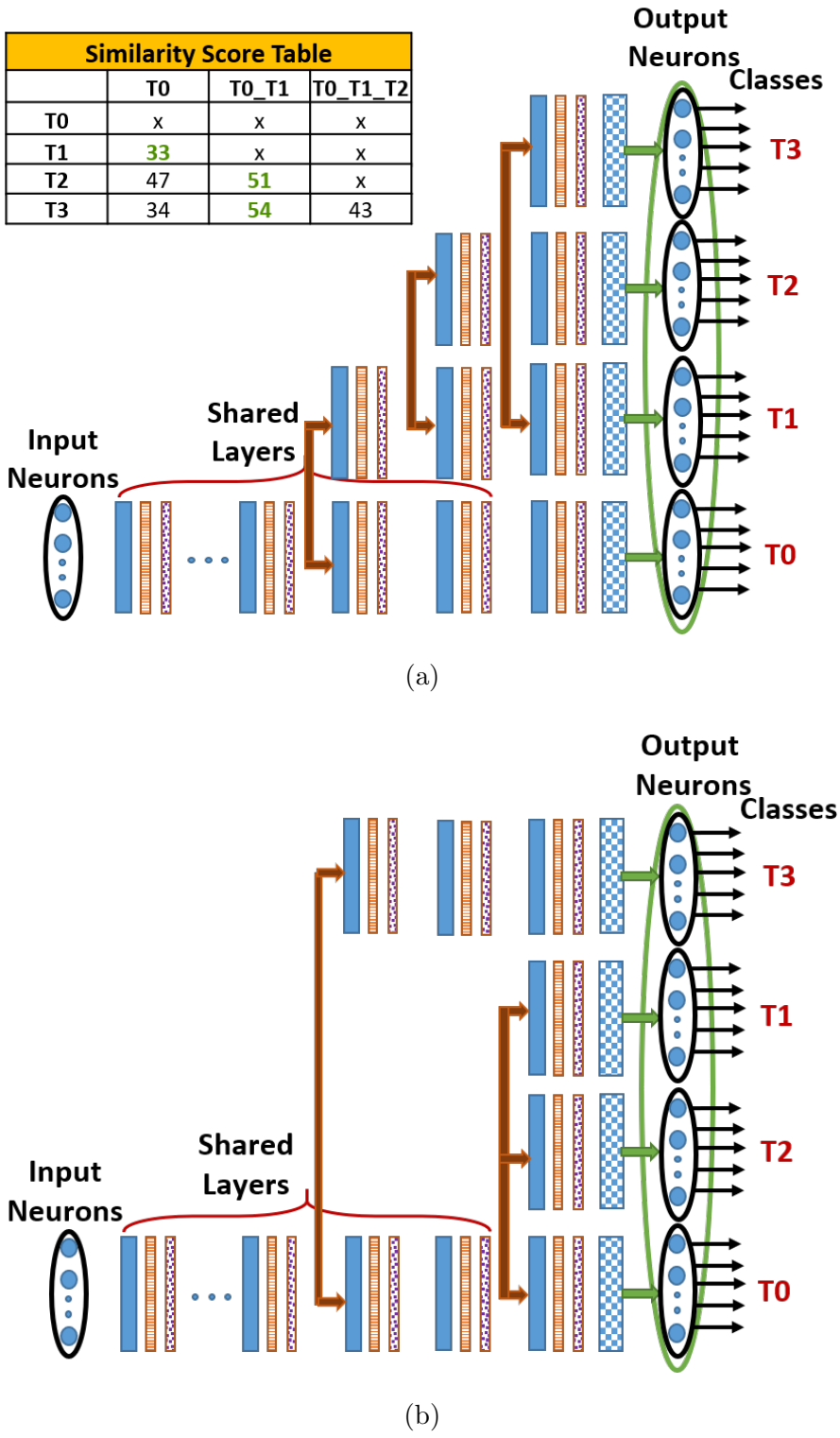


Fig. 7.12.: Updated network trained with training methodology 3 for task order (a) T0-T1-T2-T3 and (b) T0-T2-T1-T3

7.2.7 Comparison of Different Training Methodologies

Table 7.4.: Comparison of Different Training Methodologies

Task	Method 1		Method 2		Method 3	
	Accuracy	Avg. Sharing	Accuracy	Avg. Sharing	Accuracy	Avg. Sharing
T1	85.65	59.44%	86.45	46.70%	86.45	46.70%
T2	84.05	59.44%	82.30	72.20%	82.50	72.20%
T3	93.50	59.44%	93.80	46.70%	93.40	84.93%
T0-T1-T2-T3	59.51	59.44%	60.07	55.20%	60.58	67.94%

In Table 7.4, the task specific and combined classification accuracies are listed with corresponding sharing ratios. For the first training method, we used fixed sharing configuration for all new tasks. In the second training method, we utilized similarity score and found that task 1 and 3 can share less with the base network than task 2. This method reduced the average sharing while increasing the task specific and combined classification accuracy. In method 3, task 2 and task 3 were able to share higher amount with task 1 instead of task 0 while maintaining task specific performance. The combined classification is slightly improved while average sharing is also increased.

7.3 Evaluation Methodology

The proposed training approach reduces the number of kernels that would be modified during the training of new tasks. Effectively, this reduces the number of computations in the backward pass, namely layer gradients and weight gradient computations, thereby leading to energy benefits. The reduction in the number computations is an outcome of the algorithmic optimization *i.e.* reduction in the number of layers during the backpropagation for new tasks. Hence, the energy benefits are not specific to any microarchitectural feature such as dataflow, data reuse etc. In this work, we use a CMOS digital baseline based on the weight stationary dataflow

to analyze the energy consumption for DCNNs. Weight stationary has been agreed to be an efficient dataflow for executing DNN workloads [90, 91].

The baseline is a many-core architecture where each core maps a partition of the DNN. Each core is comprised of one or more Matrix Vector Multiplication (MVM) units which perform the MAC operations. An MVM unit consists of a 32KB memory module with 1024 bit bus width and 32 MACs. Thus, all the weights (32-bit weight and input) read in a single access from the local memory can be processed in the MACs in one cycle leading to a pipelined execution. Subsequently, multiple MACs within and across cores operate MVMs in parallel to execute the DNN. Note that we do not consider the energy expended in off-chip data movement (movement of weight from DRAM to local memory in cores) and inter-core data movement (over network) as these can vary based on the layer configurations, chip size, network design and several optimizations obtained from the software layers [92]. We focus only on the compute and storage energy within cores to isolate the benefits derived from the algorithmic features only. The multiplier and adder unit for MAC was implemented at the Register-Transfer Level (RTL) in Verilog and mapped to IBM 32nm technology using Synopsys Design Compiler, to obtain the energy number. The memory module in our baseline was modelled using CACTI [50], in 32nm technology library, to estimate the corresponding energy consumption.

At the algorithm level, the deep learning toolbox [45], MatConvNet [46], and PyTorch [93], which are open source neural network simulators in MATLAB, C++, and Python, are used to apply the algorithm modifications and evaluate the classification accuracy of the DCNNs under consideration. The DCNNs were trained, tested and timed using NVIDIA GPUs. In all experiments, previously seen data were not used in subsequent stages of learning, and in each case the algorithm was tested on an independent validation dataset that was not used during training. Details of the benchmarks used in our experiments are listed in Table 7.5:

Table 7.5.: Benchmarks

Application	Dataset	DCNN Structure
Character Recog.	TiCH	2 Convolutional and 1 Fully-connected Layer
Object Recog.	CIFAR-10	Network in Network [51], ResNet50 [94]
Object Recog.	CIFAR-100	ResNet18, ResNet34, ResNet50, ResNet101, DenseNet121 [95], MobileNet [96]
Object Recog.	ImageNet	ResNet34, DenseNet121, MobileNet

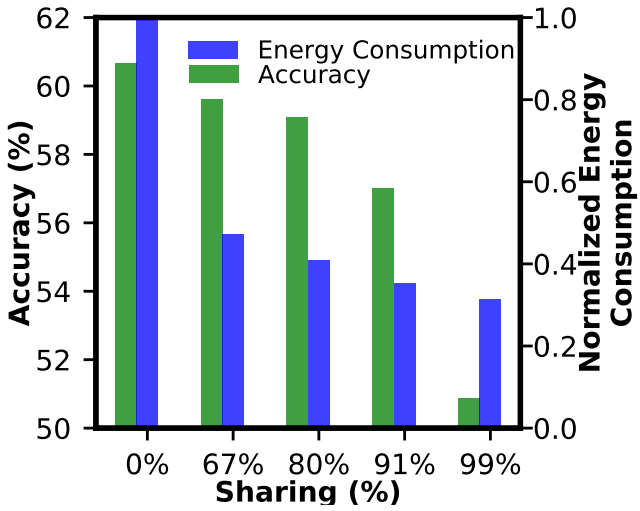
7.4 Results and Discussions

In this section, we present results that demonstrate the accuracy obtained, the energy efficiency and reduction in training time, storage requirements and memory access achieved by our proposed design. For these results, we have trained ResNet101 with CIFAR-100. The optimal sharing configuration is 80%.

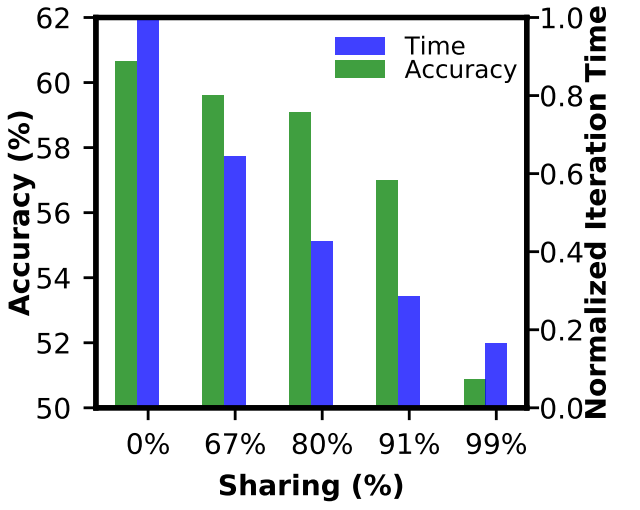
7.4.1 Energy-Accuracy Trade-off

DCNNs are trained using the standard back-propagation rule with slight modification to account for the convolutional operators [45]. The main power hungry steps of DCNN training (back-propagation) are gradient computation and weight update of the convolutional and fully connected layers [29]. In our proposed training, we achieve energy efficiency by eliminating a large portion of the gradient computation and weight update operations, with minimal loss of accuracy or output quality. The normalized energy consumption per iteration for incremental training with and without sharing convolutional layers is shown in Fig. 7.13a. The accuracies reported in this work are obtained using test datasets, which are separate from the training datasets. Based on the accuracy requirement of a specific application the optimal sharing point can be chosen from the 'Accuracy vs Sharing' curve mentioned in section 7.2.4. The optimal sharing configuration for CIFAR-100 is 80% in ResNet101. By sharing 80% of the base network parameters, we can achieve 2.45x computation energy saving while learning new tasks. The energy numbers slightly depend on number of classes in the

new tasks to be learned. However, it does not affect much since only the output layer connections vary with the number of new classes, which is insignificant compared to total connections in the network. Note that the energy mentioned in this comparison is computation energy. Memory access energy is discussed in section 7.4.3.



(a)



(b)

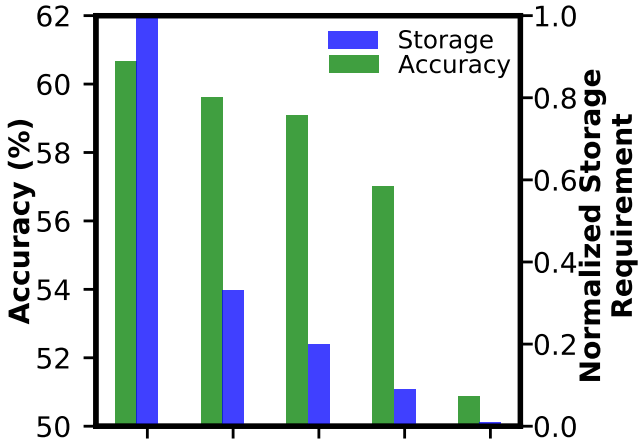
Fig. 7.13.: Comparison of (a) energy/accuracy trade-off and (b) training time requirements, between incremental training with and without sharing convolutional layers, is shown for different sharing configurations.

7.4.2 Training Time Reduction

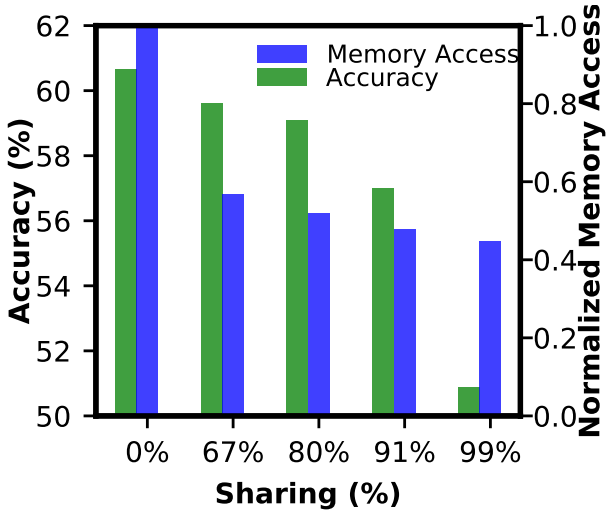
Since gradient computations and weight updates are not required for the shared convolutional layers, we achieve significant savings in computation time with our proposed approach. Fig. 7.13b shows the normalized training time per iteration for learning a set of new classes. We observe 1.55-6× reduction in training time per iteration for CIFAR-100 in ResNet101 [39] for different sharing configurations. As a byproduct of the proposed scheme, convergence becomes faster due to inheriting features from the base model. Note that the time savings cannot be used to improve accuracy of the networks by providing more epochs to the training. One way to improve accuracy is to retrain the networks with all the training samples (previously seen and unseen), which can be very time consuming and contradictory to the incremental learning principle.

7.4.3 Storage Requirement and Memory Access Reduction

Fig. 7.14a shows the storage requirement reduction obtained using our proposed scheme for CIFAR-100 in ResNet101 [39]. We achieve 67-99% reduction in storage requirement since we are sharing initial convolutional layers from the base network for the new branch networks. A large part of the training energy is spent on the memory read/write operations for the synapses. Proposed partial network sharing based training also provides 43-55% savings in memory access energy during training for CIFAR-100 in ResNet101, since we do not need to write (update during back-propagation) the fixed kernel weights during training. Fig. 7.14b shows the memory access requirement reduction obtained using proposed approach.



(a)



(b)

Fig. 7.14.: Comparison of (a) storage and (b) memory access requirements, between incremental training with and without sharing convolutional layers, is shown for different sharing configurations.

7.4.4 Results on ImageNet

The ImageNet [10] (ILSVRC2012) is one of the most challenging benchmarks for object recognition/classification. The data set has a total of ∼1.2 million la-

beled images from 1000 different categories in the training set. The validation and test set contains 50,000 and 100,000 labeled images, respectively. We implemented ResNet18, ResNet34 [39], DenseNet121 [95] and MobileNet [96], and trained them on ImageNet2012 dataset. We achieved 69.73%, 73.88%, 74.23% and 66.2% (top 1) classification accuracy for ResNet18, ResNet34, DenseNet121 and MobileNet, respectively, in regular training (all 1000 classes trained together), which are the upper-bounds for the incremental learning on corresponding networks. Then we divided the dataset into 3 sets of 500, 300 and 200 classes for the purpose of incremental learning. The classes for forming the sets were chosen randomly and each set was mutually exclusive. The set of 500 classes were used to train the base network. The other two sets were used for incremental learning. Utilizing our proposed method, we obtained the optimal sharing configuration. For ResNet18, we were able to share only $\sim$1.5% of the learning parameters from the base network and achieve classification accuracy within $1 \pm 0.5\%$ of the baseline (network w/o sharing) accuracy. On the other hand, using ResNet34, we were able to share up to 33% of the learning parameters from the base network. The classification accuracy results for ResNet34 with $\sim$33% sharing configuration are listed in Table 7.6. This implies that the amount of network sharing largely depends on the network size and architecture. For instance, the DenseNet121 with 121 layers provides $\sim$57% sharing, while the MobileNet with only 25 layers provides up to 34% sharing of the learning parameters (for similar accuracy specifications on ImageNet dataset). Combined classification accuracy achieved on DenseNet121 and MobileNet are $\sim$64% and $\sim$62%, respectively. The following sub-section analyses the performance and corresponding benefits of proposed methodology on different network architectures.

7.4.5 Comparison between Different Network Architectures

We observed that the optimal network sharing configuration depends on network architecture. Therefore, careful consideration is required while selecting the network

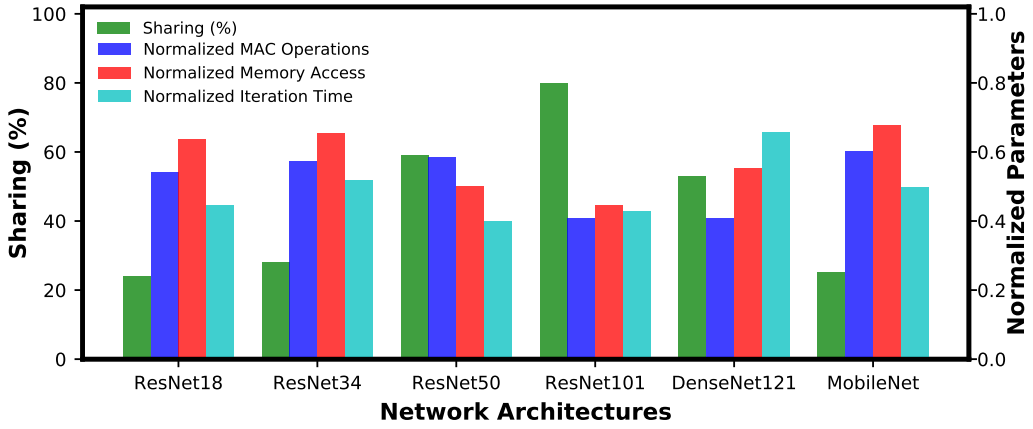
Table 7.6.: Accuracy results for ResNet34 trained on ImageNet

#Classes	Accuracy(%) w/o sharing		Accuracy(%) w/ sharing	
	Top 1%	Top 5%	Top 1%	Top 5%
1000 (all classes)	73.88	91.70	-	-
500	80.85	95.37	-	-
300	74.3	90.67	71.25	89.2
200	75.83	92.61	76.6	93.12
1000 (updated)	66.99	87.4	65.85	86.65

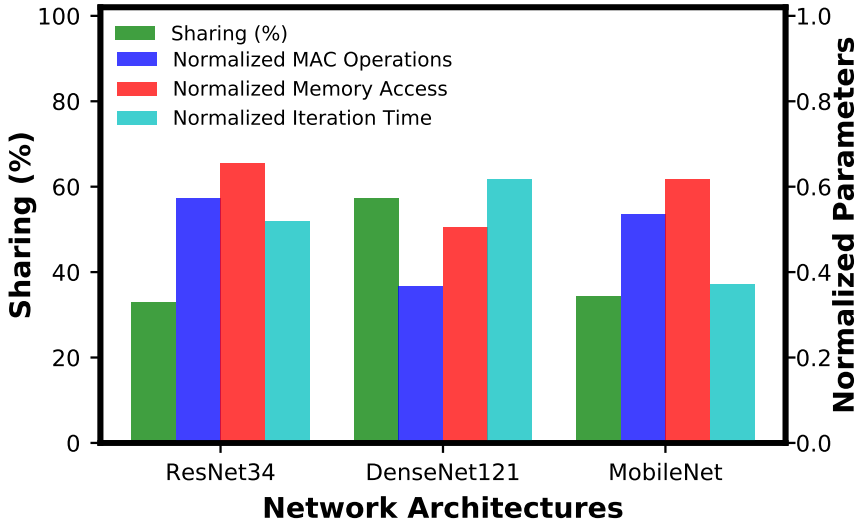
for using proposed methodology. We experimented with ResNet networks of different depths, DenseNet [95] and MobileNet [96]. The networks are trained on CIFAR-100 and ImageNet with class divisions in Table 7.3 and 7.6. For these experiments, we used $1 \pm 0.5\%$ accuracy degradation as a tolerance value for determining the optimal sharing configuration. Fig. 7.15a shows a comparison between four ResNets (18, 34, 50, 101), Densenet121 and MobileNet network trained with CIFAR-100, while Fig. 7.15b shows a comparison between three networks (ResNet34, DenseNet121 and MobileNet) trained for ImageNet dataset. We observed that deeper networks (>30 layers) provide more energy benefits for minimal accuracy degradation. However, sharing ratio does not have a linear relation with the energy benefits and training time savings. In Fig. 7.15, we can observe that for similar sharing configurations, different networks achieve different amount of reduction in computational energy, memory access and iteration time. For instance, sharing $\sim$33% of the learning parameters in ResNet34, we reduced training time per iteration by 51% (Fig. 7.15b). On the other hand, similar amount of sharing in MobileNet reduces training time per iteration by 62% (Fig. 7.15b).

For both CIFAR-100 and ImageNet dataset, MobileNet performs similar to ResNet34 in terms of parameter sharing and energy benefits, while DenseNet provides higher amount of parameter sharing and energy benefits since it has much more depth. The

trend confirms that there are two prime conditions which need to be satisfied for getting superior performance using the proposed algorithm. Firstly, the network has to be deep enough so that enough layers from base network can be shared. Small networks have most of the weights in the final FC layers which cannot be shared. Larger networks allow more learning parameters to be shared without performance loss. For instance, we could share a lot more of the learning parameters from the base network in ResNet34 compared to ResNet18 for ImageNet. Also for CIFAR-100, percentage sharing is very high in ResNet101 (80%) compared to ResNet18 (24%) for an equivalent accuracy degradation. It can also be seen that a small network (Table 7.2) does not show efficient feature sharing as it is not deep enough. We did experiments with CIFAR-10 using ResNet50 and achieved better results ($\sim$74% accuracy on combined classification, compared to $\sim$59% in Table 7.2) with $\sim$48% network sharing. Secondly, the base network must contain a good number of features. The combined network performance is best for ImageNet (Table 7.6) among other datasets and much closer to the cumulatively trained network (which is the theoretical upper-bound). This is due to the fact that in the case of ImageNet, the base network has learned sufficient features since it was trained with large number of classes and examples. On the other hand, accuracy is worst for CIFAR-10 (Table 7.2) as its base network learns only 4 classes and has significantly lower number of training samples.



(a)



(b)

Fig. 7.15.: Comparison between different network architectures trained on (a) CIFAR-100 and (b) ImageNet, using proposed algorithm. For these experiments, we used 1±0.5% accuracy degradation as a tolerance value for determining the optimal sharing configuration.

Table 7.7.: Qualitative Comparison with Other Methods

Performance Metric	Fine Tuning	Feature Extraction	Cumulative Training	Learning w/o Forgetting	This Work
New Task Accuracy	best	medium	best	best	good
Old Task Accuracy	worst	good	best	good	best
Training Speed	fast	fast	slow	fast	fastest
Inference Speed	fast	fast	fast	fast	fast
Previous Data Required	no	no	yes	no	no
Storage Requirement	low	medium	highest	low	medium

7.4.6 Comparison with Other Methods

Table 7.7 presents a qualitative comparison between different methods for incremental learning namely; fine tuning, feature extraction, joint training, Learning without Forgetting [81] and our proposed method. Here, we considered that accuracy for each task (old and new) is measured separately as reported in [81]. In fine tuning, the entire network is retrained for a few epochs to learn the new task. It suffers from catastrophic forgetting and forgets the old task since old task data is not used during retraining. In feature extraction, a trained feature extractor is used to extract features for the new task and then a separate classifier is trained on the extracted features. Although it does not forget the old task, new task performance is lower since the feature extractor is not explicitly trained to extract appropriate features for the new task. Cumulative training achieves the best accuracy on new tasks without forgetting the old tasks. However, it requires old task samples to be stored, which leads to a higher memory requirement compared to the other approaches. Also cumulative training is slower compared to other approaches since every retraining utilizes all data samples from old and new tasks. 'Learning without Forgetting' (LwF) fine tunes the network for new tasks while maintaining response of the new task samples on old task specific neurons [81] . It achieves higher performance on the new task compared to other approaches since the entire network is fine tuned for the new task. LwF training is fast as it fine tunes the network with new samples only compared to training from scratch using all the task samples. LwF adds the least number of parameters for the

new task [81], thereby enabling fast inference and lower memory requirements. LwF aims to achieve energy efficient inference while learning new task with tolerable loss in old task accuracy. However, a key drawback of LwF is the partial forgetting of old task(s) during the fine-tuning process (learning new-task). For combined classification, LwF suffers significant accuracy drop. For instance, ResNet32 incrementally trained in two steps each having 50 classes of CIFAR-100 obtains a top 1 accuracy of ∼52.5% with LwF [82]. In a similar setup, our proposed training obtains 62.1% while reducing the computation energy (∼60%), memory access (∼48%) and iteration time (∼57%) in training. Even with periodical utilization of old data samples, [82] reaches 62% (top 1) accuracy. [82] also shows that with classes learned in more than two installments, LwF may suffer from continuously degrading performance on old tasks. On the contrary, we focus on achieving energy efficient training (for new tasks) without tolerating any accuracy degradation in old tasks. To this effect, we consider scenarios where learning the new task needs to be efficient. Consequently, we trade minimal accuracy while achieving maximal energy benefit for the new task. Our proposed methodology does not alter the task specific parameters (branch network) for old tasks as well as the shared parameters during the learning of new tasks, thereby not degrading performance for old tasks. Further, for the new task we only need to fine-tune the small cloned branch, which makes the training faster compared to other methods.

Task-specific inference cost (energy and time) remains similar for the proposed approach compared to the baseline, as we can activate the specific branch only. The combined classification (input can belong to any task) cost will grow sub-linearly as we add more branches, since several branches share initial layer computations. An alternative approach to learn new task(s) while retaining the previous task-accuracy would require training separate networks with no weight sharing. Subsequently, the inference cost will be maximum in the case of combined classification, as all separate networks would have to be evaluated. Hence, higher sharing improves the efficiency for both training and inference.

In a typical scenario, a model is used for a lot of inference tasks once it is (re-)trained. Nonetheless, we focus on facilitating energy constraint training. With the advent of emerging algorithms and technology, such scenario will become very popular in near future if on-chip training is made energy-efficient. For example, cell phones are increasingly employing on-chip facial and finger print recognition. In addition to that, on-chip learning will alleviate the requirements to send the data to cloud, thereby enhancing security. Similarly, drones can be employed to learn new tasks on the fly without storing the data samples due to memory constraints. In such applications, proposed training algorithm can be beneficial.

Next, we will quantitatively compare our approach of incremental learning with two standard approaches in Fig. 7.16 for ImageNet dataset. The standard approaches are:

1. Cumulative: In this approach the model is retrained with the samples of the new classes and all the previous learned classes (all previous data must be available). This is sort of an upper bound for ideal incremental learning. Since in incremental learning, the old data samples are not available, it will remain an open problem until an approach can match performance of the cumulative approach without using stored data for already learned classes.

2. Naïve: In this approach, the network is completely retrained with the data samples of new classes only. It suffers from catastrophic forgetting. This approach is also termed as 'Fine tuning'.

From Fig. 7.16, we can observe that the performance of our proposed method is not very far from the cumulative approach. We also observe that our proposed partial network sharing approach performs almost same as the approach without partial network sharing. While the Naïve approach always performs well for the new set of classes only since the network forgets the old classes due to catastrophic forgetting.

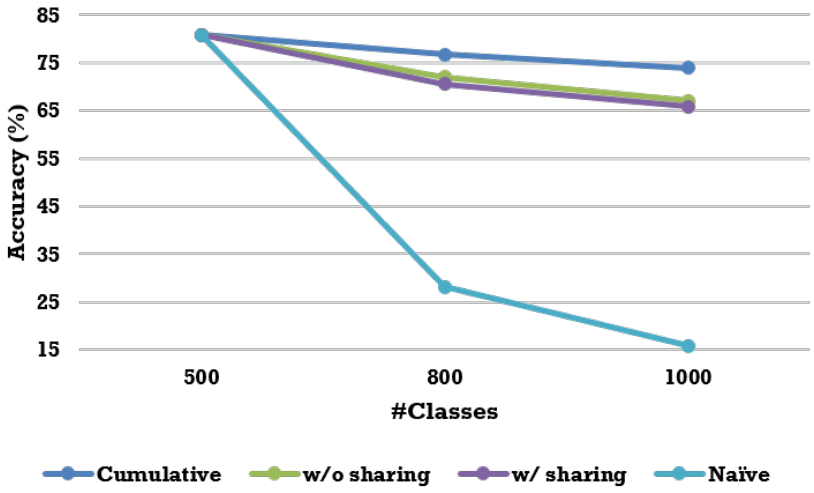


Fig. 7.16.: Performance comparison of incremental learning approaches.

8. ENABLING SPIKE-BASED BACKPROPAGATION IN STATE-OF-THE-ART DEEP SPIKING NEURAL NETWORK ARCHITECTURES

Over the last few years, deep learning has made tremendous progress and has become a prevalent tool for coping with various cognitive tasks such as object detection, speech recognition and reasoning. Various deep learning techniques [8, 40, 97] enable the effective optimization of deep ANNs by constructing multiple levels of feature hierarchies and show remarkable results, which occasionally outperform human level performance [9, 13, 39]. To that effect, deploying deep learning is becoming necessary not only on large-scale computers, but also on edge devices (*e.g.* phone, tablet, smart watch, robot etc.). However, the ever-growing complexity of the state-of-the-art deep neural networks together with the explosion in the amount of data to be processed, place significant energy demands on current computing platforms. For example, a deep ANN model requires unprecedented amount of computing hardware resources that often requires huge computing power of cloud servers and significant amount of time to train.

Spiking Neural Networks (SNNs) are the leading candidates for overcoming the constraints of neural computing and to efficiently harness the machine learning algorithm in real-life (or mobile) applications [98, 99]. The concepts of SNN, which is often regarded as the 3^{rd} generation neural network [100], are inspired by biologically plausible Leaky Integrate and Fire (LIF) spiking neuron models [101] that can efficiently process spatio-temporal information. The LIF neuron model is characterized by the internal state, called membrane potential, that integrates the inputs over time and generates an output spike (or Dirac delta pulse) whenever it reaches the neuronal firing threshold. This mechanism enables event-driven and asynchronous computations across the layers on spiking systems, which makes it naturally suitable

for ultra-low power and low latency operation. Furthermore, recent works [102, 103] have shown that these properties make SNNs significantly more attractive for deeper networks in the case of hardware implementation. This is because the spike signals become significantly sparser as the layer goes deeper, such that the number of required computations significantly reduces. In this context, several training strategies can be applied to take full advantage of SNNs.

The general training strategy of SNNs can be categorized in two ways - ANN-SNN conversion and direct spike-based training. First, there are studies which have successfully deployed the ANN-SNN conversion technique that transforms off-line trained ANN to SNN for efficient event-driven inference [102–106]. The main objective of ANN-SNN conversion scheme is to leverage the state-of-the-art ANN training techniques, so that the transformed networks can mimic the competitive classification performances of the ANNs. For instance, specialized SNN hardwares (such as SpiNNaker [107], IBM TrueNorth [98]) have exhibited greatly improved power efficiency as well as the state-of-the-art performance for inferencing. However, it takes large number of time-steps (latency) to resemble the input-output mapping of pre-trained ANN counterpart. This is because, only Integrate-and-Fire (IF) spiking neuron can be replaced with an ANN (ReLU) neuron, and hence, can not effectively capture the temporal dynamics of spatio-temporal event-driven information. On the other hand, it is still a difficult problem to directly train a deep spiking neural network using input spike events and spike-based learning algorithm, mainly because of discontinuous and non-differentiable spike generation function and discrete nature of spike events. To that effect, unsupervised Spike-Timing-Dependent-Plasticity (STDP) learning algorithm has been explored for training two-layer SNNs (consisting of input and output layers) by considering the local correlations of pre- and post- neuronal spike timing. STDP-trained two-layer network (consisting of 6400 output neurons) has been shown to achieve 95% classification accuracy on MNIST dataset. However, shallow network structure limits the expressive power of neural network [108–112] and suffers from scalability issues as the classification performance easily saturates. Layer-wise

STDP learning [113, 114] has shown the capabilities of efficient feature extraction on multi-layer convolutional SNNs. Nevertheless, the performance gaps compared to ANN models (trained with standard BP algorithm) are still significantly large. The unsatisfactory classification performances of unsupervised local learning necessitate a spike-based supervised learning rule such as gradient descent backpropagation (BP) algorithm [7]. In the context of SNNs, the spike-based BP learning algorithm introduced in [115,116] dealt with the membrane potential as a differentiable activation of spiking neuron to train the synaptic weights. [117] apply BP based supervised training for the classifier after training the feature extractor layer by layer using auto-encoder mechanism. By leveraging the best of both unsupervised and supervised learning, [118] have shown that layer-wise STDP learning along with spike-based BP have synergistic effect to improve the robustness, generalization ability as well as acceleration of training speed. In this work, we take these prior works forward to effectively train very deep SNNs using end-to-end spike-based gradient descent backpropagation learning.

The main contributions of our work are specified as follows. First, we develop a spike-based supervised gradient descent BP algorithm that exploits a conditionally differentiable approximated activation function of LIF neuron. In addition, we leverage the key idea of the successful deep ANN models such as LeNet5 [8], VGG [38] and ResNet [39] for efficiently constructing state-of-the-art deep SNN network architectures. We also adapt dropout [40] technique in order to better regularize deep SNN training. Next, we demonstrate the effectiveness of our methodology for visual recognition tasks on standard character and object datasets (MNIST, SVHN, CIFAR-10) and a neuromorphic dataset (N-MNIST). To the best of our knowledge, this work achieves the best classification accuracy in MNIST, SVHN and CIFAR-10 datasets through training deep SNNs. Lastly, we expand our efforts to quantify and analyze the advantages of spike-based BP algorithm compared to ANN-SNN conversion techniques in terms of inference time and energy consumption.

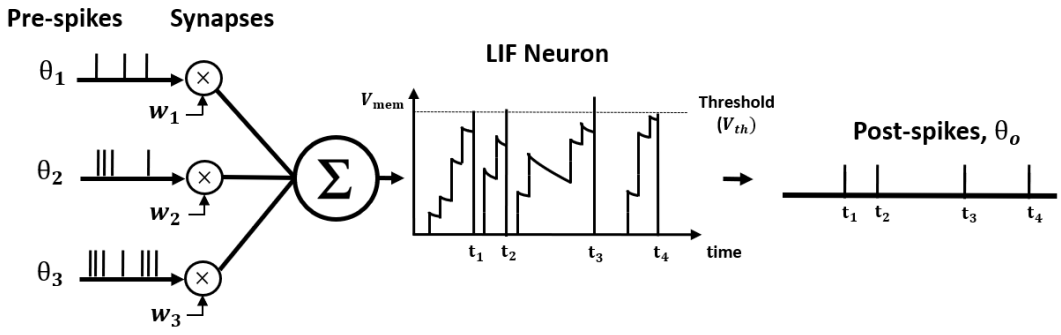


Fig. 8.1.: The operation of a Leaky Integrate and Fire (LIF) neuron.

8.1 The Components and Architecture of Spiking Neural Network

8.1.1 Spiking Neural Network Components

The Leaky-Integrate-and-Fire (LIF) neurons [101] and plastic synapses are fundamental and biologically plausible computational elements for emulating the dynamics of SNNs. The neurons in adjacent layers are massively inter-connected via each associated plastic synapse whereas no connection exists within a layer. The spike input signals always move in one direction, a way from the input layer through the hidden layers and to the output layer. The dynamics of LIF spiking neuron can be formulated as:

$$\tau \frac{dV_{mem}}{dt} = -V_{mem} + \sum_{i=1}^{n^l} (w_i * \theta_i(t - t_k)) \tag{1}$$

where V_{mem} means post-neuronal membrane potential, τ is the time constant for membrane potential decay, n^l indicates the number of pre-neurons, w_i is the synaptic weight connecting i^{th} pre-neuron to post-neuron and $\theta_i(t - t_k)$ denotes a spike event from i^{th} pre-neuron at time t_k. The operation of a LIF neuron is presented in Fig. 8.1. The impacts of each pre-spike, $\theta_i(t-t_k)$, are modulated by the corresponding synaptic weight (w_i) to generate the current influx flowing into the post-neuron in the next layer. The stimulus fed as current influx is integrated in the post-neuronal membrane

potential (V_{mem}) that leaks exponentially over time. The decay constant (τ) decides the degree of membrane leakage over time and a smaller value of τ indicates stronger membrane potential decay. When the accumulated membrane potential reaches or exceeds a certain neuronal firing threshold (V_{th}), the corresponding neuron generates a post-spike to the fan-out synapses and resets its own membrane potential to initial value (zero). In Table 8.1, we list the annotations used in equations (1-14).

Table 8.1.: List of Notations

Notations	**Meaning**
θ	Spike
x	Sum of spike events throughout the time
w	Synaptic weight
V_{mem}	Membrane potential
V_{th}	Neuronal firing threshold
net	Total (incoming) current influx throughout the time
a	Activation of spiking neuron
E	Loss function
δ	Error gradient

8.1.2 Deep Convolutional Spiking Neural Network

Building Blocks

In this work, we develop a training methodology for convolutional SNN models that consist of an input layer followed by intermediate hidden layers and a final output layer. In the input layer, the pixel images are encoded as Poisson-distributed spike trains, where the probability of spike generation is proportional to the pixel intensity. The hidden layers consist of multiple convolutional (C) and spatial-pooling (P)

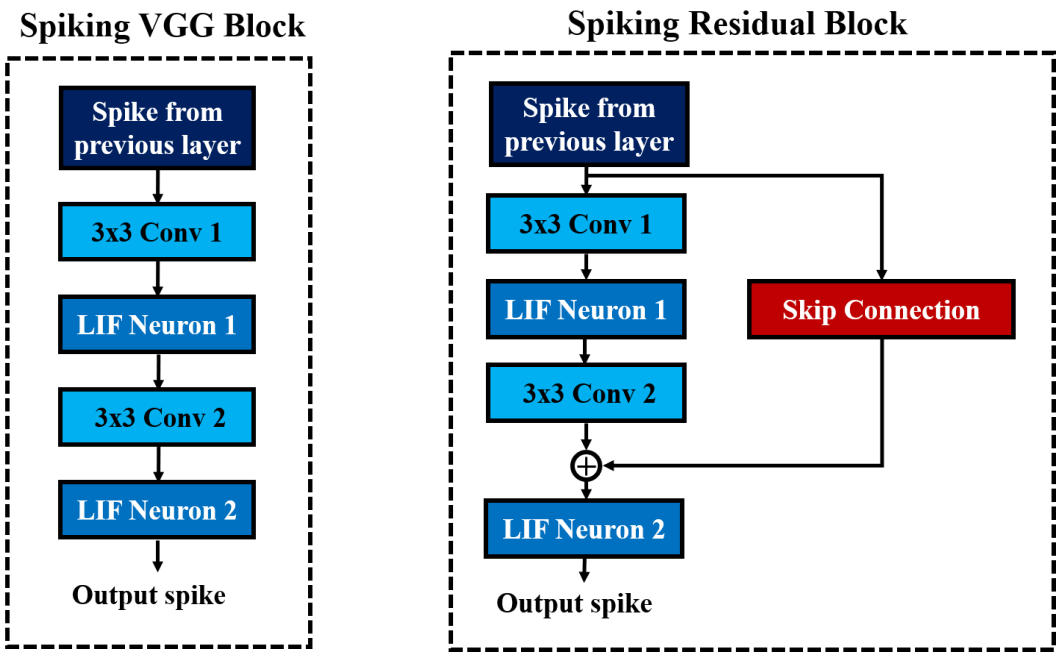


Fig. 8.2.: Basic building blocks of (a) VGG and (b) ResNet architectures in deep convolutional SNNs.

layers which are often arranged in an alternating manner. These convolutional (C) and spatial-pooling (P) layers represent the intermediate stages of feature extractor. The spikes from the feature extractor are combined to generate one dimensional vector input for the fully-connected (FC) layers to produce the final classification. The convolutional and fully-connected layers contain trainable parameters (*i.e.* synaptic weights), while the spatial-pooling layers are fixed *a priori*. Through the training procedure, weight kernels in the convolutional layers can encode the feature representations of the input patterns at multiple hierarchical levels. Therefore, through convolution operation, the trained convolutional kernels can detect the spatially correlated local features in the input patterns. This inherently allows the network to be invariant to translation (shift) in the object location. A convolutional layer is often followed by a spatial-pooling layer. The spatial-pooling layer is used to downscale the

dimensions of the feature maps, produced by the previous convolutional layer, while retaining the spatial correlation between neighborhood pixels in every feature map.

There are various choices for performing the spatial-pooling operation in the ANN domain. The two major choices are max-pooling (maximum neuron output over the pooling window) or average-pooling (two-dimensional average pooling operation over the pooling window). In most of the state-of-the-art deep ANNs, max-pooling is considered as the most popular option. However, since the neuron activations are binary in SNNs instead of analog values, max-pooling does not provide useful information to the following layer. Therefore, we have used averaging mechanism for spatial-pooling. In SNNs, average-pooling scheme is different than in ANN as an additional thresholding is used after averaging to generate output spikes. For instance, a fixed 2×2 kernel (each having a weight of 0.25) strides through a convolutional feature map without overlapping and fires an output spike at the corresponding location in the pooled feature map only if the sum of the weighted spikes of the 4 inputs within the kernel window exceeds a designated threshold. The threshold for average-pooling has to be carefully set, so that the spike propagation is not disrupted due to the pooling. If the threshold is too low, then there will be too many spikes that can cause loss of spatial location of the feature that was extracted from the previous layer. On the other hand, if the threshold is too high, then there will not be enough spike propagation to the deeper layers. We have used a threshold of 0.75 for a fixed (2×2) kernel (each having a weight of 0.25) in the average pooling layers. This means that if there are at least 3 spikes in the (2×2) window, then 1 spike will be generated in the pooled map. For a different kernel size, the threshold has to be properly adjusted maintaining a similar ratio (0.75). The pooling operation provides several key benefits. First, it reduces size of the convolutional feature maps and provides additional network invariance to input transformations. Furthermore, the pooling operation enlarges the effective size of convolutional kernels in the following layer as the feature maps are downscaled. This allows consecutive convolutional layers to efficiently learn hierarchical representations from low to high levels of abstractions. The number of pooled feature maps is the

same as the number of output feature maps of the previous convolutional layer. The feature maps of the final pooling layer before the fully-connected layers are unrolled into a 1-D vector to be used as input for a fully-connected layer. There are one or more fully-connected layers eventually reaching to the output layer which produces inference decisions. This final fully-connected part of the network acts as a classifier to effectively incorporate the composition of features resulting from the alternating convolutional and pooling layers into the final output classes.

Deep Convolutional SNN architecture: VGG and Residual SNNs

Deep network topologies are essential for recognizing complex input patterns so that they can effectively learn hierarchical representations. To that effect, we investigate the state-of-the-art deep neural network architectures such as VGG [38] and ResNet [39] in order to build deep SNN architectures. VGG [38] was one of the first neural networks which used the idea of using small (3×3) convolutional kernels uniformly throughout the network. The utilization of small (3×3) kernels enables effective stacking of convolutional layers while minimizing the number of parameters in deep networks. In this work, we build deep convolutional SNNs (containing more than 5 trainable layers) by using *'Spiking VGG Block'* which contains stack of convolutional layers using small (3×3) kernels. The Fig. 8.2a shows a *'Spiking VGG block'* containing two stacked convolutional layers with intermediate LIF neuronal layer. Next, ResNet [39] introduced the skip connections throughout the network that had large success in enabling successful training of significantly deeper networks. In particular, ResNet addresses the degradation (of training accuracy) problem [39] that occurs while increasing the number of layers in normal feedforward neural network. We employ the concept of the skip connection to construct deep residual SNNs whose number of trainable layers is 7-11. The Fig. 8.2b shows a 'Spiking Residual Block' consisting of non-residual and residual paths. The non-residual path consists of two convolutional layers with an intermediate LIF neuronal layer. The residual

path (skip connection) is composed of the identity mapping when the number of input and output feature maps are the same, and 1×1 convolutional kernels when the number of input and output feature maps are different. Both of the non-residual and residual path outputs are integrated to the membrane potential in the last LIF neuronal layer (LIF Neuron 2 in Fig. 8.2b) to generate output spikes from the 'Spiking Residual Block'. Within the feature extractor, a 'Spiking VGG Block' or 'Spiking Residual Block' is often followed by an average-pooling layer to construct the alternating convolutional and spatial-pooling structure. Note, in some 'Spiking Residual Blocks', last convolutional and residual connections employ convolution with stride of 2 to incorporate the functionality of the spatial-pooling layers. At the end of the feature extractor, extracted features from the last average-pooling layer is fed to a fully-connected layer as a 1-D vector input for initiating the classifier operation.

8.2 Supervised Training of Deep Spiking Neural Network

8.2.1 Spike-based Gradient Descent Backpropagation Algorithm

The spike-based BP algorithm in SNN is adapted from standard BP [7] in the ANN domain. In standard BP, the network parameters are iteratively updated in a direction to minimize the difference between the final outputs of the network and target labels. The standard BP algorithm achieves this goal by back-propagating the output error through the hidden layers using gradient descent method. However, the major difference between ANNs and SNNs is the dynamics of neuronal output. An artificial neuron (such as *sigmoid*, *tanh*, or $ReLU$) communicates via continuous values whereas a spiking neuron generates binary spike outputs over time. In SNNs, spatio-temporal spike trains are fed to the network as inputs. Accordingly, the outputs of spiking neuron are spike events which are discontinuous and discrete (non-differentiable) over time. Hence, the standard BP algorithm is incompatible to train SNNs, as it can not back-propagate the gradient through non-differentiable neuronal functions. We formulate the a differentiable (but approximated) activation of LIF

neuron that enables modulation of the network parameters using gradient descent method in spiking system. The spike-based BP can be divided into three phases - forward propagation, backward propagation and weight update. We now describe the spike-based BP algorithm by going through each phase.

Forward Propagation

In forward propagation, spike trains representing input patterns and corresponding output (target) labels are presented to the network. To generate the spike inputs, the input pixel values are converted to Poisson-distributed spike trains and delivered to the network. The input spikes are multiplied with synaptic weights to produce an input current. The resultant current is accumulated in the membrane potential of post neurons as indicated by equation (1). The post-neuron generates an output spike whenever the respective membrane potential exceeds a neuronal firing threshold. Otherwise, membrane potential decays exponentially with time. After the post-neuronal firing, the membrane potential is reset, and the output spike is broadcast to be the input to the subsequent layer. The post-neurons of every layer carry out this process successively based on the weighted spikes received from the preceding layer. Over time, the total weighted summations of the spike trains are integrated at the j^{th} post-neuron as formulated in equation (2). The sum of spike trains (denoted by $x_i(t)$ for the i^{th} input neuron) is weighted by inter-connecting synaptic weights, w_{ij}.

$$net_j^{l+1}(t) = \sum_{i=1}^{n^l}(w_{ij}^l * x_i(t)),\ where\ x_i(t) = \sum_{k=1}^{t}\theta_i(t-t_k) \tag{2}$$

where $net_j^{l+1}(t)$ stands for the total (resultant) current influx received by j^{th} post-neuron throughout the time t, n^l is the number of pre-neurons and $\theta_i(t-t_k)$ is a spike event from i^{th} pre-neuron at time instant t_k.

However, the neuronal firing threshold of the final layer is set to a very high value such that the output neurons do not generate any spike output. In the final layer, the weighted spikes from previous layer are accumulated in the membrane potential while

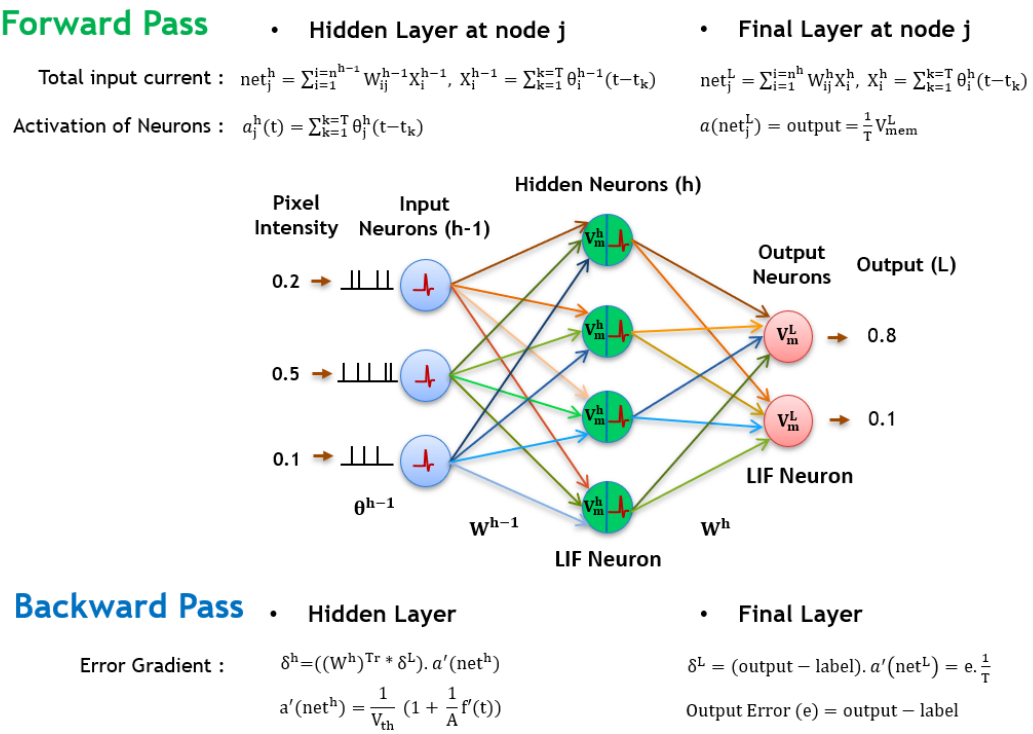


Fig. 8.3.: Illustration the two phases (forward propagation and backward propagation) of spike-based backpropagation algorithm in a LIF neuron.

decaying over time. At the last time step, the accumulated membrane potential is divided by the number of total time steps in order to quantify the output distribution (*output*) as presented by equation (3).

$$output = \frac{V_{mem}^{L}}{number\ of\ timesteps} \tag{3}$$

Backward Propagation and Weight Update

Next, we formulate the gradient-based backward propagation [7] for SNNs. The first step is to estimate the gradients of loss function at the output layer. The loss function is a measure of discrepancy between target labels and outputs predicted by the network. Then, the gradients are propagated backward all the way down to the inputs through the hidden layers using recursive chain rule (equation 4). The

following equations (4-14) and Fig. 8.3 describe the detailed mathematical steps for obtaining the partial derivatives of error with respect to weights.

The prediction error of each output neuron is evaluated by comparing the output distribution (*output*) with the desired target label (*label*) of the presented input spike trains as shown in equation (5). The corresponding loss function (E in equation (6)) is defined as the sum of squared (final prediction) error over all the output neurons. To calculate the $\frac{\partial E}{\partial a}$ and $\frac{\partial a}{\partial net}$ terms, we need to define the differentiable activation function of LIF neuron. In SNN, the 'activation function' indicates the relationship between weighted summation of pre-neuronal spike inputs and post-neuronal outputs over time. For the output layer, we can use the output value from equation (3) as activation since it is a continuous variable. Hence, $\frac{\partial E}{\partial a}$ is equal to the final output error as calculated in equation (7). Moreover, we consider the leak in the membrane potential of the final layer neurons as noise so that the accumulated membrane potential is approximated as equivalent to the (total) net input current ($V_{mem}^{L} \approx net$). Therefore, the derivative of post-neuronal activation with respect to net input current ($\frac{\partial a}{\partial net}$) is calculated as $\frac{1}{T}$ for the final layer. However, for the hidden layers, we have spike trains as outputs. A spike output signal is non-differentiable since it is discrete and creates a discontinuity (because of step jump) at the time instance of firing. Therefore, we need to have an activation function that can be differentiated to apply the chain rule. To get around this predicament, we define an 'conditionally differentiable approximate activation' by low-pass filtering the individual post-spikes as formulated in equation (8). To compute the activation, $f(t)$, of a LIF neuron, the unit spikes (at time instants t_k) are temporally integrated and the resultant sum is decayed within the time periods. The time constant (τ) determines the decay rate of the spiking neuronal activation. It influences the temporal dynamics of the spiking neuron by accounting for the exponential membrane potential decay and reset mechanisms. It is evident that $f(t)$ is continuous except for the time points where spikes occur and the activities jump up [116]. Hence, $f(t)$ is differentiable at $t \to t_k^+$. Note that, to capture the leaky

effect (exponential decay), it is necessary to compute derivative of $f(t)$ at points in between the spiking activities, not at the time of spiking.

$$\frac{\partial E}{\partial w^l} = \frac{\partial E}{\partial a}\frac{\partial a}{\partial net}\frac{\partial net}{\partial w^l} \tag{4}$$

$$Final\ output\ error,\ e_j = output_j - label_j \tag{5}$$

$$Loss\ function,\ E = \frac{1}{2}\sum_{j=1}^{n^L} {e_j}^2 \tag{6}$$

$$\frac{\partial E}{\partial a} = \frac{\partial}{\partial output}\ \frac{1}{2}(output - label)^2 = output - label = e \tag{7}$$

$$Activation\ of\ neuron,\ f(t) = \sum_{k=1}^{t} exp(-\frac{t - t_k}{\tau}) \tag{8}$$

$$\frac{\partial a}{\partial net} = a'(net) = \frac{1}{V_{th}}(1 + \frac{1}{A}f'(t)) = \frac{1}{V_{th}}(1 + \frac{1}{A}\sum_{k=1}^{t} -\frac{1}{\tau}e^{-\frac{t-t_k}{\tau}}) \tag{9}$$

To obtain a derivative for LIF neuronal activation with respect to net input current, we take help of several approximations. We first estimate the derivative of an 'Integrate and Fire' (IF) neuron's activation. Then, with the derivative of IF neuron's activation, we add the slope of the scaled LIF neuron activation ($f(t)$), to account for the leak effect in the membrane potential. Since IF neuron activation is a step function, we approximate it as a linear function with slope of unity value (assuming the firing threshold is set to 1), using straight through estimation concept [119]. In equation (9), the unity represents the approximate derivative for IF neuron activation. Due to the leaky nature of a LIF neuron, $f(t)$ has a negative slope measured at time instance when the neuron is not spiking ($t \to t_k^+$). Hence, the combination of straight through estimation (approximate derivative of IF neuron) and time derivative of approximate activation (scaled) of LIF neuron is less than 1. So far, we have assumed that the firing threshold for both IF and LIF activation is set to unity value. However, if the firing threshold is set to a low (high) value, the frequency of neuronal firing would increase (decrease). Hence, the estimation of derivative for neuronal activation is normalized by neuronal threshold in equation (9) to reflect the inverse relationship.

In summary, the three approximations applied to implement backpropagation in SNN are as follows:

- We consider the leak in the membrane potential of the final (output) layer neurons as noise so that the accumulated membrane potential is approximated as equivalent to the (total) net input current ($V_{mem}^{L} \approx net$). Therefore, the derivative of post-neuronal activation with respect to net input current ($\frac{\partial a}{\partial net}$) is calculated as $\frac{1}{T}$ for the final layer.

- For hidden layers, we first approximate the activation of an IF neuron as a linear function. Hence, we are able to estimate its derivative of IF neuron's activation [119] with respect to net input current.

- To capture the leaky effect of a LIF neuron (in hidden layers), we estimate the scaled time derivative of the low pass filtered output spikes that leak over time, using the function $f(t)$ (equation 8). This function is continuous except for the time points where spikes occur [116]. Hence, it is differentiable in the sections between the spiking activities.

- We obtain a pseudo derivative for LIF neuronal activation (in hidden layers) as a combination of two derivatives. The first one is the derivative of IF neuron's activation with respect to net input current. The second one is the derivative (at $t \to t_k^+$) of $f(t)$ with respect to time.

Based on these approximations, we have build a framework that can train SNNs from direct spike inputs using backpropagation.

$$\delta^{L} = \frac{\partial E}{\partial a}\frac{\partial a}{\partial net} = e.\frac{1}{T} = \frac{e}{T} \tag{10}$$

$$\delta^{h} = ((w^{h})^{Tr} * \delta^{h+1}).a'(net^{h}) \tag{11}$$

At the output layer, the error gradient, δ^{L}, represents the gradient of the output loss with respect to the net input current received by the post-neurons. It can be

calculated by multiplying the final output error (e) with the derivative of the corresponding post-neuronal activation, $a'(net^L)$, with respect to its inputs as shown in equation (10). Note that element-wise multiplication is indicated by . while matrix multiplication is represented by * in the respective equations. At any hidden layer, the local error gradient, δ^h, is recursively estimated by multiplying the back-propagated gradient from the successive layer ($(w^h)^{Tr} * \delta^{h+1}$) with derivative of the neuronal activation ($a'(net^h)$) as presented in equation (11).

$$\frac{\partial net}{\partial w^l} = \frac{\partial}{\partial w^l}(w^l * \sum_{k=1}^{t} \theta^l(t - t_k)) = \sum_{k=1}^{t} \theta^l(t - t_k) \tag{12}$$

$$\triangle w^l = \frac{\partial E}{\partial w^l} = (\sum_{k=1}^{t} \theta^l(t - t_k)) * (\delta^{l+1})^{Tr} \tag{13}$$

$$w^l_{updated} = w^l - \eta_{BP} \triangle w^l \tag{14}$$

The derivative of net current with respect to weight is simply the total incoming spikes throughout the time as derived in equation (12). The derivative of the output loss with respect to the weights interconnecting the layers l and $l+1$ ($\triangle w^l$ in equation (13)) is determined by multiplying the transposed error gradient at $l + 1$ (δ^{l+1}) with the input spikes from layer l. Finally, the calculated partial derivatives of loss function are used to update the respective weights using a learning rate (η_{BP}) as illustrated in equation (14). As a result, iterative updating of the weights over mini-batches of input patterns leads the network state to a local minimum, thereby enabling the network to capture multiple-levels of internal representations of the data.

8.2.2 Dropout in Spiking Nerual Network

Dropout [40] is one of the popular regularization techniques while training deep ANNs. This technique randomly disconnects certain units with a given probability (p) to avoid units being overfitted and co-adapted too much to given training data. We employ the concept of dropout technique in order to effectively regularize deep SNNs. Note, dropout technique is only applied during training and is not used when

evaluating the performance of the network through inference. There is a subtle difference in the way dropout is applied in SNNs compared to ANNs. In ANNs, each epoch of training has several iterations of mini-batches. In each iteration, randomly selected units (with dropout ratio of p) are disconnected from the network while weighting by its posterior probability ($\frac{1}{1-p}$). However, in SNNs, each iteration has more than one forward propagation depending on the time length of the spike train. We back-propagate the output error and modify the network parameters only at the last time step. For dropout to be effective in our training method, it has to be ensured that the set of connected units within an iteration of mini-batch data is not changed, such that the neural network is constituted by the same random subset of units during each forward propagation within a single iteration. On the other hand, if the units are randomly connected at each time-step, the effect of dropout will be averaged out over the entire forward propagation times within an iteration. Then, the dropout effect would fade-out once the output error is propagated backward and the parameters are updated at the last time step. Therefore, it is necessary to keep the set of randomly connected units for entire time window within an iteration. In the experiment, we use the SNN version of dropout technique with the probability (p) of omitting units equal to 0.2-0.25. Note that the activation is much sparser in SNN forward propagation compared to ANN, hence the optimal p for SNNs need to be less than typical ANN dropout ratio (p=0.5). The details of SNN forward propagation with dropout are specified in Algorithm 4.

8.3 Experimental Setup

The primary goal of our experiments is to demonstrate the effectiveness of the proposed spike-based BP training methodology in a variety of deep network architectures. We first describe our experimental setup and baselines. For the experiments, we developed a custom simulation framework using the Pytorch [93] deep learning package for evaluating our proposed SNN training algorithm. Our deep convolutional

ALGORITHM 4: Forward propagation with Dropout at each iteration in SNN

1. **Input** : Poisson input spike train ($inputs$), Dropout ratio (p), Total number of time steps ($\#timesteps$)
2. // Define the random subset of units (with a probability $1-p$) at each iteration
3. **for** $i \leftarrow 1$ to $\#net.layer$ **do**
4. $\quad mask[i] = generate\_random\_subset(probability = 1-p)$
5. **for** $t \leftarrow 1$ to $\#timesteps$ **do**
6. $\quad$ // Set input of first layer equal to spike train of a mini-batch data
7. $\quad net.layer[1].spike[t] = inputs;$
8. $\quad$ **for** $i \leftarrow 2$ to $\#net.layer$ **do**
9. $\quad\quad$ // Integrate weighted sum of input spikes to membrane potential with decay over time
10. $\quad\quad net.layer[i].v_{mem} = net.layer[i].v_{mem} * e^{-\frac{1}{T_p}} + net.layer[i] : forward(net.layer[i].spike[t]) .* mask[i] / (1-p);$
11. $\quad\quad$ // Post-neuron fires if membrane potential is greater than neuronal threshold
12. $\quad\quad net.layer[i+1].spike[t+1] = net.layer[i].v_{mem} > net.layer[i].v_{th}$
13. $\quad\quad$ // Reset the membrane potential if post-neuron fires
14. $\quad\quad net.layer[i].v_{mem}(net.layer[i+1].spike[t+1]) = 0$

SNNs are populated with biologically plausible LIF neurons in which a pair of pre- and post- neurons are interconnected by plastic synapses. At the beginning, the neuronal firing thresholds are set to an unity value and the synaptic weights are initialized with Gaussian random distribution of zero-mean and standard deviation of $\sqrt{\frac{\alpha}{n^l}}$ (n^l: number of fan-in synapses) as introduced in [120]. Note, the initialization constant α differs by the type of network architecture. For instance, we have used $\alpha = 2$ for non-residual network and $\alpha = 1$ for residual network. For training, the synaptic weights are trained with mini-batch spike-based BP algorithm in an end-to-end manner as explained in section 8.2.1. For static datasets, we train our network models for 150 epochs using mini-batch stochastic gradient descent BP that reduces its learning rate at 70^{th}, 100^{th} and 125^{th} training epoch. For the neuromorphic dataset, we use

Adam [121] learning method and reduce its learnig rate at 40^{th}, 80^{th} and 120^{th} training epoch. Please, refer to Table 8.2 for more implementation details. The datasets and network topologies used for benchmarking, the spike generation scheme for event driven operation and determination of the number of time-steps required for training and inference are described in the following sub-sections.

Table 8.2.: Parameters used in the Experiments

Parameter	**Value**
Decay Constant of Membrane Potential and Neuronal Activation (τ)	100 time-steps
BP Training Time Duration	50-100 time-steps
Inference Time Duration	Same as training
Mini-batch Size	16-32
Spatial-pooling Non-overlapping Region/Stride	2×2, 2
Weight Initialization Constant (α)	2 (non-residual network), 1 (residual network)
Learning rate (η_{BP})	0.002 - 0.003
Dropout Ratio (p)	0.2 - 0.25

Table 8.3.: Benchmark Datasets

Dataset	Image	#Training Samples	#Testing Samples	#Category
MNIST	28×28, gray	60,000	10,000	10
SVHN	28×28, color	73,000	26,000	10
CIFAR-10	32×32, color	50,000	10,000	10
N-MNIST	$34 \times 34 \times 32$, ON and OFF spikes	60,000	10,000	10

8.3.1 Benchmarking Datasets

We demonstrate the efficacy of our proposed training methodology for deep convolutional SNNs on three standard vision datasets and one neuromorphic vision dataset, namely the MNIST [8], SVHN [122], CIFAR-10 [69] and N-MNIST [123]. The MNIST

dataset is composed of gray-scale (one-dimensional) images of handwritten digits whose sizes are 28 by 28. The SVHN and CIFAR-10 datasets are composed of color (three-dimensional) images whose sizes are 32 by 32. The N-MNIST dataset is a neuromorphic (spiking) dataset which is converted from static MNIST dataset using Dynamic Vision Sensor (DVS) [124]. The N-MNIST dataset contains two-dimensional images that include ON and OFF event stream data whose sizes are 34 by 34. The ON (OFF) event represents the increase (decrease) in pixel brightness. Details of the benchmark datasets are listed in Table 8.3. For evaluation, we report the top-1 classification accuracy by classifying the test samples (training samples and test samples are mutually exclusive).

8.3.2 Network Topologies

We use various SNN architectures depending on the complexity of the benchmark datasets. For MNIST and N-MNIST datasets, we used a network consisting of two sets of alternating convolutional and spatial-pooling layers followed by two fully-connected layers. This network architecture is derived from LeNet5 model [8]. Note that Table 8.4 summarizes the layer type, kernel size, the number of output feature maps and stride of SNN model for MNIST dataset. The kernel size shown in the table is for 3-D convolution where the 1^{st} dimension is for number of input feature-maps and 2^{nd}-3^{rd} dimensions are for convolutional kernels. For SVHN and CIFAR-10 datasets, we used deeper network models consisting of 7 to 11 trainable layers including convolutional, spatial-pooling and fully-connected layers. In particular, these networks consisting of 5 or more trainable layers are constructed using small (3×3) convolutional kernels. We term the deep convolutional SNN architecture that includes 3×3 convolutional kernel [38] without residual connections as 'VGG SNN' and with skip (residual) connections [39] as 'Residual SNN'. In Residual SNNs, some convolutional layers convolve kernel with stride of 2 in both x and y directions, to incorporate the functionality of spatial-pooling layers. Please, refer to Tables 8.4 and

8.5 that summarize the details of deep convolutional SNN architectures. In the results section, we will discuss the benefit of deep SNNs in terms of classification performance as well as inference speedup and energy efficiency.

Table 8.4.: The deep convolutional spiking neural network architectures for MNIST, N-MNIST and SVHN dataset

4 layer network				**VGG7**				**ResNet7**			
Layer type	**Kernel size**	**#o/p feature-maps**	**Stride**	**Layer type**	**Kernel size**	**#o/p feature-maps**	**Stride**	**Layer type**	**Kernel size**	**#o/p feature-maps**	**Stride**
Convolution	1×5×5	20	1	Convolution	3×3×3	64	1	Convolution	3×3×3	64	1
Average-pooling	2×2		2	Convolution	64×3×3	64	2	Average-pooling	2×2		2
				Average-pooling	2×2		2				
Convolution	20×5×5	50	1	Convolution	64×3×3	128	1	Convolution	64×3×3	128	1
Average-pooling	2×2		2	Convolution	128×3×3	128	2	Convolution	128×3×3	128	2
				Convolution	128×3×3	128	2	Skip convolution	64×1×1	128	2
				Average-pooling	2×2		2				
								Convolution	128×3×3	256	1
								Convolution	256×3×3	256	2
								Skip convolution	128×1×1	256	2
Fully-connected		200		Fully-connected		1024		Fully-connected		1024	
Output		10		Output		10		Output		10	

Table 8.5.: The deep convolutional spiking neural network architectures for a CIFAR-10 dataset

VGG9				**ResNet9**				**ResNet11**			
Layer type	**Kernel size**	**#o/p feature-maps**	**Stride**	**Layer type**	**Kernel size**	**#o/p feature-maps**	**Stride**	**Layer type**	**Kernel size**	**#o/p feature-maps**	**Stride**
Convolution	3×3×3	64	1	Convolution	3×3×3	64	1	Convolution	3×3×3	64	1
Convolution	64×3×3	64	1	Average-pooling	2×2		2	Average-pooling	2×2		2
Average-pooling	2×2		2								
Convolution	64×3×3	128	1	Convolution	64×3×3	128	1	Convolution	64×3×3	128	1
Convolution	128×3×3	128	1	Convolution	128×3×3	128	1	Convolution	128×3×3	128	1
Average-pooling	2×2		2	Skip convolution	64×1×1	128	1	Skip convolution	64×1×1	128	1
Convolution	128×3×3	256	1	Convolution	128×3×3	256	1	Convolution	128×3×3	256	1
Convolution	256×3×3	256	1	Convolution	256×3×3	256	2	Convolution	256×3×3	256	2
Convolution	256×3×3	256	1	Skip connection	128×1×1	256	2	Skip convolution	128×1×1	256	2
Average-pooling	2×2		2								
				Convolution	256×3×3	512	1	Convolution	256×3×3	512	1
				Convolution	512×3×3	512	2	Convolution	512×3×3	512	1
				Skip convolution	256×1×1	512	2	Skip convolution	512×1×1	512	1
								Convolution	512×3×3	512	1
								Convolution	512×3×3	512	2
								Skip convolution	512×1×1	512	2
Fully-connected		1024		Fully-connected		1024		Fully-connected		1024	
Output		10		Output		10		Output		10	

8.3.3 ANN-SNN Conversion Scheme

As mentioned previously, off-line trained ANNs can be successfully converted to SNNs by replacing ANN (ReLU) neurons with Integrate and Fire (IF) spiking neurons and adjusting the neuronal thresholds with respect to synaptic weights. It is important to set the neuronal firing thresholds sufficiently high so that each spiking neuron can closely resemble ANN activation without loss of information. In the literature, several methods have been proposed [102–106] for balancing appropriate ratios between neuronal thresholds and synaptic weights of spiking neuron in the case of ANN-SNN conversion. In this work, we compare various aspects of our direct-spike trained models with one recent work [102], which proposed a near-lossless ANN-SNN conversion scheme for deep network architectures. In brief, [102] balanced the neuronal firing thresholds with respect to corresponding synaptic weights layer-by-layer depending on the actual spiking activities of each layer using a subset of training samples. Basically, we compare our direct-spike trained model with converted SNN on the same network architecture in terms of accuracy, inference speed and energy-efficiency. Please note that there are couple of differences on the network architecture and conversion technique between [102] and our scheme. First, [102] always uses average-pooling to reduce the size of previous convolutional output feature-map, whereas our models interchangeably use average pooling or convolve kernels with stride of 2 in convolutional layer. Next, [102] only consider identity skip connections for residual SNNs. However, we implement skip connections using either identity mapping or 1×1 convolutional kernel. Lastly, we used lower (0.75) threshold for avg-pooling layer instead of 0.8 to ensure enough spike propagation on both direct-trained and converted network models. Even in the case of ANN-SNN conversion scheme, lower average-pooling threshold provides us slightly better classification performance than [102].

8.3.4 Spike Generation Scheme

For the static vision datasets (MNIST, SVHN and CIFAR-10), each input pixel intensity is converted to stream of Poisson distributed spike events that have equivalent firing rates. The Poisson input spikes are fed to the network over time. This rate-based spike encoding is used for a given period of time during both training and inference. For color image datasets, we use image pre-processing techniques of random cropping and horizontal flip before generating input spikes. These input pixels are normalized to represent zero mean and unit standard deviation. Thereafter, we scale the pixel intensities to bound them in the range [-1,1] to represent the whole spectrum of input pixel representations. The normalized pixel intensities are converted to Poisson spike events such that the generated input signals are bipolar spikes. For the neuromorphic version of dataset (N-MNIST), we use the original (unfiltered and uncentered) version of spike streams to directly train and test the network in time domain.

8.3.5 Time-steps

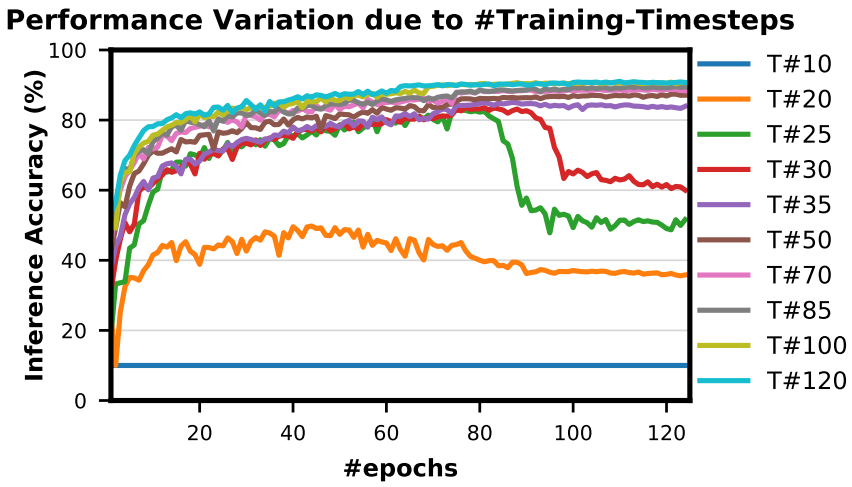
As mentioned in section 8.3.4, we generate stochastic Poisson spike train for each input pixel intensity for event-driven operation. The duration of the spike train is very important for SNNs. We measure the length of the spike train (spike time window) in time-steps. For example, a 100 time-step spike train will have approximately 50 random spikes if the corresponding pixel intensity is half in a range of [0,1]. If the number of time-steps (spike time window) is too less, then the SNN will not receive enough information for training or inference. On the other hand, if the number of time-steps is too high, then the latency will also be high and the spike stream will behave more like a deterministic input. Hence, the stochastic property of SNNs will be lost, the inference will become too slow, and the network will not have much energy efficiency over ANN implementation. For these reasons, we experimented with different number of time-steps to empirically obtain the optimal number of time-steps

required for both training and inference. The experimental process and results are explained in the following subsections.

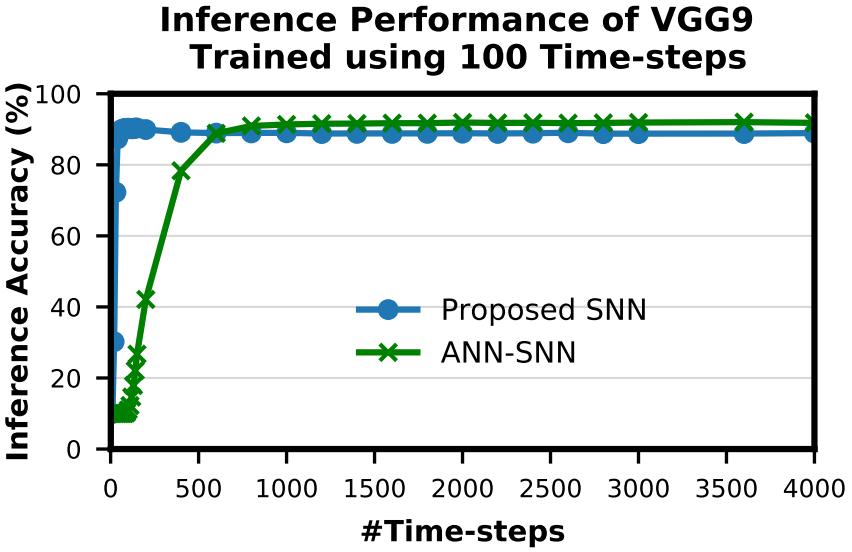
Optimal #time-steps for Training

A spike event can only represent 0 or 1 in each time step, therefore usually its bit precision is considered 1. However, the spike train provides temporal data, which is an additional source of information. Therefore, the spike train length (number of time-steps) in SNN can be considered as its actual precision of neuronal activation. To obtain the optimal #time-steps required for our proposed training method, we trained a VGG9 network on CIFAR-10 dataset using different time-steps ranging from 10 to 120 (shown in Fig. 8.4a). We found that for only 10 time-steps, the network is unable to learn anything as there is not enough information (input precision too low) for the network to be able to learn. This phenomena is explained by the lack of spikes in the final output. With the initial weights, the accumulated sum of the LIF neuron is not enough to generate output spikes in the latter layers. Hence, none of the input spikes propagates to the final output neurons and the output distributions remain 0. Therefore, the computed gradients are always 0 and the network is not updated. For 20-30 time-steps, some input spikes are able to reach the final layer, hence the network starts to learn but do not converge. For 35-50 time-steps, the network learns well and converges to a reasonable point. From 70 time-steps, the network accuracy starts to saturate. At about 100 time-steps the network training improvement completely saturates. This is consistent with the bit precision of the inputs. It has been shown in [35] that 8 bit inputs and activations are sufficient to achieve optimal network performance for standard image recognition tasks. Ideally, we need 128 time-steps to represent 8 bit inputs using bipolar spikes. However, 100 time-steps proved to be sufficient as more time-steps provide marginal improvement. We observe similar trend in VGG7, ResNet7, ResNet9 and ResNet11 SNNs as well, while training for SVHN and CIFAR-10 datasets. Therefore, we considered 100 time-

steps as the optimal #time-steps for training in our proposed methodology. Moreover, for MNIST dataset, we used 50 time-steps since the required bit precision is only 4 bits [35].



(a)



(b)

Fig. 8.4.: Inference performance variation due to (a) #Training-Timesteps and (b) #Inference-Timesteps. $T\#$ in (a) indicates number of time-steps used for training.

Optimal #time-steps for Inference

To obtain the optimal #time-steps required for inferring an image utilizing a network trained with our proposed method, we conducted similar experiments as described in section 8.3.5. We first trained a VGG9 network for CIFAR-10 dataset using 100 time-steps (optimal according to experiments in section 8.3.5). Then, we tested the network performances with different time-steps ranging from 10 to 4000 (shown in Fig. 8.4b). We observed that the network performs very well even with only 10 time-steps, while the peak performance occurs around 100 time-steps. For more than 100 time-steps, the accuracy degrades slightly from the peak. This behavior is very different from ANN-SNN converted networks where the accuracy keeps on improving as #time-steps is increased (shown in Fig. 8.4b). This can be attributed to the fact that our proposed spike-based training method incorporates the temporal information well in to the network training procedure so that the trained network is tailored to perform best at a specific spike time window when inferencing. On the other hand, the ANN-SNN conversion schemes are unable to incorporate the temporal information of the input in the trained network and therefore are heavily dependent on the deterministic behavior of the input. Hence, the ANN-SNN conversion schemes require much higher #time-steps for inference in order to resemble input-output mappings similar to ANNs.

8.4 Results

In this section, we analyze the classification performance and efficiency achieved by the proposed spike-based training methodology for deep convolutional SNNs compared to the performance of the transformed SNN using ANN-SNN conversion scheme.

8.4.1 The Classification Performance

Most of the classification performances available in literature for SNNs are for MNIST, N-MNIST and CIFAR-10 datasets. The popular methods for SNN training are 'Spike Time Dependent Plasticity (STDP)' based unsupervised learning [108–112] and 'Spike-based Backpropagation' based supervised learning [116, 125–128]. There are a few works [113,118,129,130] which tried to combine the two approaches to get the best of both worlds. However, these training methods were able to neither train deep SNNs nor achieve good inference performance compared to ANN implementations. Hence, ANN-SNN conversion schemes have been explored by researchers [102–106]. Till date, ANN-SNN conversion schemes achieved the best inference performance for CIFAR-10 dataset using deep networks [102, 103]. Classification performances of all these works are listed in Table 8.6 along with ours. To the best of our knowledge, we achieved the best inference accuracy for MNIST using LeNet structured network. We also achieved accuracy performance comparable with ANN-SNN converted network [102] for CIFAR-10 dataset using much smaller network models, while beating all other SNN training methods.

For a more extensive comparison, we compare inference performances of trained networks using our proposed methodology with the state-of-the-art ANNs and ANN-SNN conversion scheme, for same network configuration (depth and structure) side by side in Table 8.7. We also compare with the previous best SNN training results found in literature that may or may not have same network depth and structure as ours. The ANN-SNN conversion scheme is a modified and improved version of [102]. We are using this modified scheme since it achieves better conversion performance than [102] as explained in section 8.3.3. Note that all reported classification accuracies are the average of the maximum inference accuracies for three independent runs with different seeds.

After initializing the weights, we train the SNNs using spike-based BP algorithm in an end-to-end manner with Poisson spike train inputs. Our evaluation on a MNIST

Table 8.6.: Comparison of the SNNs classification accuracies on MNIST, N-MNIST and CIFAR-10 datasets.

Model	**Learning Method**	**Accuracy (MNIST)**	**Accuracy (N-MNIST)**	**Accuracy (CIFAR-10)**
Hunsberger *et al.* [106]	Offline learning, conversion	98.37%	–	82.95%
Esser *et al.* [131]	Offline learning, conversion	–	–	89.32%
Diehl *et al.* [105]	Offline learning, conversion	99.10%	–	–
Rueckauer *et al.* [103]	Offline learning, conversion	99.44%	–	88.82%
Sengupta *et al.* [102]	Offline learning, conversion	–	–	91.55%
Kheradpisheh *et al.* [113]	Layerwise STDP + offline SVM classifier	98.40%	–	–
Panda *et al.* [117]	Spike-based autoencoder	99.08%	–	70.16%
Lee *et al.* [116]	Spike-based BP	99.31%	98.74%	–
Wu *et al.* [126]	Spike-based BP	99.42%	98.78%	50.70%
Lee *et al.* [118]	STDP-based pretraining + spike-based BP	99.28%	–	–
Jin *et al.* [125]	Spike-based BP	99.49%	98.88%	–
Wu *et al.* [132]	Spike-based BP	–	99.53%	90.53%
This work	Spike-based BP	99.59%	99.09%	90.95%

Table 8.7.: Comparison of Classification Performance

Inference Accuracy (%)					
Dataset	Model	ANN	ANN-SNN	SNN [Previous Best]	SNN [This Work]
MNIST	LeNet	99.57	99.59	99.49 [125]	99.59
N-MNIST	LeNet	–	–	99.53 [132]	99.09
SVHN	VGG7	96.36	96.30	–	96.06
	ResNet7	96.43	95.93	–	96.21
CIFAR-10	VGG9	91.98	92.01	90.53 [132]	90.45
	ResNet9	91.85	89.00		90.35
	ResNet11	91.87	90.15		90.95

dataset yields a classification accuracy of 99.59% which is the best compared to any other SNN training scheme and also our ANN-SNN conversion scheme. We achieve ∼96% inference accuracy on SVHN dataset for both trained non-residual and resid-

ual SNN which is very close to the state-of-the-art ANN implementation. Inference performance for SNNs trained on SVHN dataset have not been reported previously in literature.

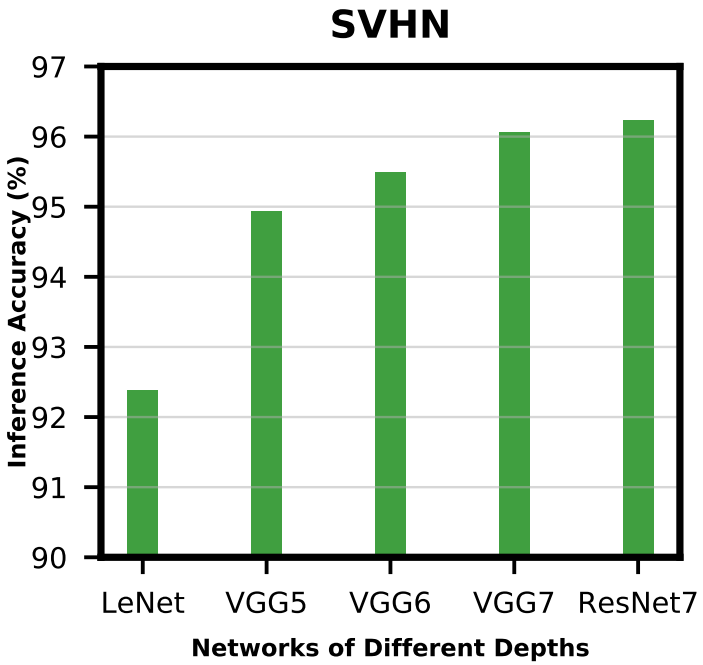
We implemented three different networks, as shown in Table 8.5, for classifying CIFAR-10 dataset using proposed spike-based BP algorithm. For the VGG9 network, the ANN-SNN conversion scheme provides near lossless converted network compared to baseline ANN implementation, while our proposed training method yields a classification accuracy of 90.45%. For ResNet9 network, the ANN-SNN conversion scheme provides inference accuracy within 3% of baseline ANN implementation. However, our proposed spike-based training method achieve better inference accuracy that is within $\sim$1.5% of baseline ANN implementation. In the case of ResNet11, we observe that the inference accuracy improvement is marginal compared to ResNet9 for baseline ANN implementation. However, ANN-SNN conversion scheme and proposed SNN training show improvement of $\sim$0.5% for ResNet11 compared to ResNet9. Overall, for ResNet networks, our proposed training method achieves better inference accuracy compared to ANN-SNN conversion scheme.

8.4.2 Accuracy Improvement with Network Depth

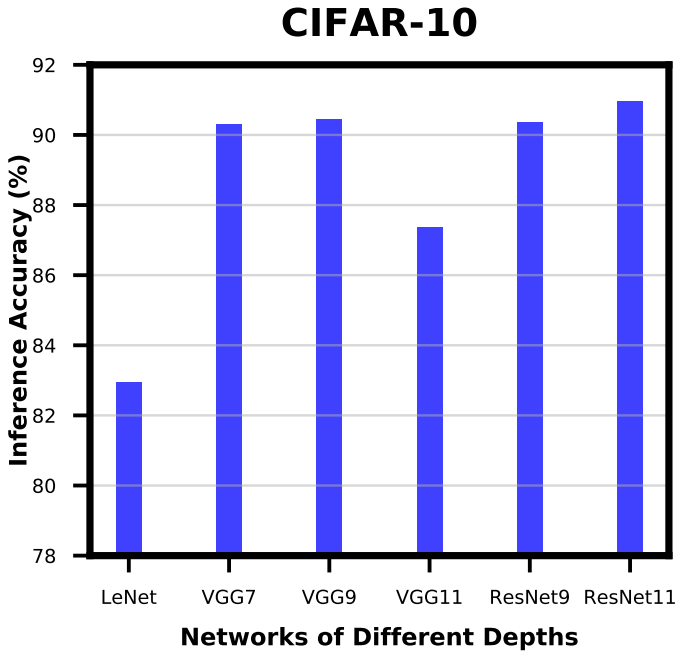
One of the major drawbacks of STDP based unsupervised learning for SNNs is that it is very difficult to train beyond 2 convolutional layers [113, 114]. Therefore, researchers are leaning more towards gradient-based (BP) supervised learning for deep SNNs.

In order to analyze the effect of network depth for direct-spike trained SNNs, we experimented with networks of different depths while training for SVHN and CIFAR-10 datasets. For SVHN dataset, we started with a small network derived from LeNet5 model [8] with 2 convolutional and 2 fully-connected layers. This network was able to achieve inference accuracy of only 92.38%. Then, we increased the network depth by adding 1 convolutional layer before the 2 fully-connected layers and we termed this

network as VGG5. VGG5 network was able to achieve significant improvement over its predecessor. Similarly, we tried VGG6 followed by VGG7, and the improvement started to become very small. We have also trained ResNet7 to understand how residual networks perform compared to non-residual networks of similar depth. Results of these experiments are shown in Fig. 8.5a. We carried out similar experiments for CIFAR-10 dataset as well. The results show similar trend (Fig. 8.5b). These results ensure that network depth improves learning capacity of direct-spike trained SNNs similar to ANNs. The non-residual networks saturate at certain depth and start to degrade if network depth is further increased (VGG11 in Fig. 8.5b) due to the degradation problem mentioned in [39]. In such scenario, the residual connections in deep residual ANNs allows the network to maintain peak classification accuracy utilizing the skip connections [39] as seen in Fig. 8.5b (ResNet9 and ResNet11).



(a)



(b)

Fig. 8.5.: Accuracy Improvement with Network Depth for (a) SVHN dataset and (b) CIFAR-10 dataset.

8.5 Discussion

8.5.1 Comparison with Relevant works

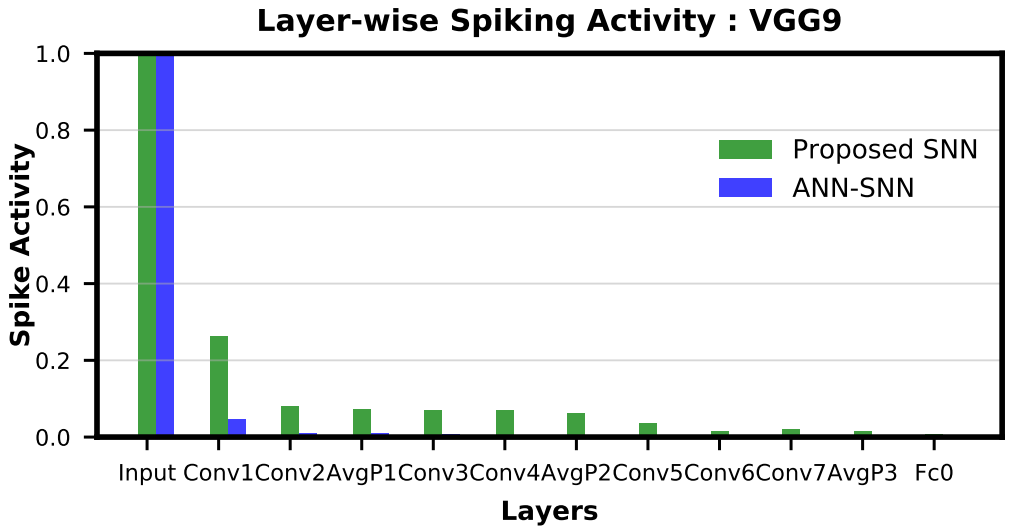
In this section, we compare our proposed supervised learning algorithm with other recent spike-based BP algorithms. The spike-based learning rules primarily focus on directly training and testing SNNs with spike-trains and no conversion is necessary for applying in real-world spiking scenario. In recent years, there is a significant increase in number of supervised gradient descent methods in spike-based learning. [117] developed spike-based auto-encoder mechanism to train deep convolutional SNNs. They dealt with membrane potential as a differentiable signal and showed recognition capabilities in standard vision tasks (MNIST and CIFAR-10 datasets). [116] followed the similar approach to explore a spike-based BP algorithm in an end-to-end manner. In addition, [116] presented the error normalization scheme to prevent exploding gradient phenomenon for training deep SNNs. [125] proposed hybrid macro/micro level backpropagation (HM2-BP). HM2-BP is developed to capture the temporal effect of individual spike (in micro-level) and rate-encoded error (in macro-level). In temporal encoding domain, [128] proposed an interesting temporal spike-based BP algorithm by treating the spike-time as the differential activation of neuron. Temporal encoding based SNN has the potential to process spatio-temporal spike patterns with small number of spikes. All of these works demonstrated spike-based learning in simple network architectures and has large gap in classification accuracy compared to deep ANNs. More recently, [132] presented a neuron normalization technique (called NeuNorm) that calculates the average input firing rates to adjust neuron selectivity. NeuNorm enables spike-based training within relatively short time-window while achieving competitive performances. In addition, they presented an input encoding scheme that receives both spike and non-spike signals for preserving the precision of input data.

There are several points that distinguish our work from the others. First, we derived a conditionally differentiable (but approximated) activation of a LIF neuron

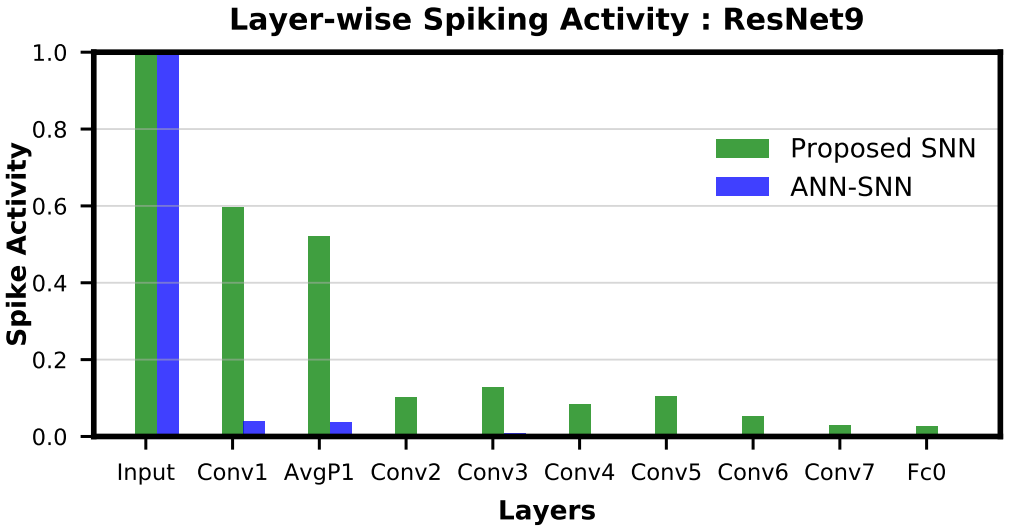
given the measured neuronal outputs (as defined in equation (3)). The activation of a LIF neuron is formulated as 'low-pass filtered output signal' which is the accumulation of leaky output spikes throughout the time. In back-propagating phase, the defined activation of a LIF neuron enables us to calculate the neuronal pseudo-derivative while accounting for the leaky behavior (as explained in equation (9)). Note that the effect of leaky component has high impact on the dynamics of LIF spiking neuron. It is worth mentioning here that the better approximation of LIF neuronal activation function enables our network to achieve better performance than the other methods in the literature. Next, we construct our networks by leveraging state-of-the-art deep architectures such as VGG [38] and ResNet [39]. To the best of our knowledge, this is the first work that demonstrates spike-based supervised BP learning for SNNs containing more than 10 trainable layers. Our deep SNNs obtain the superior classification accuracies in MNIST, SVHN and CIFAR-10 datasets in comparison to the other networks trained with spike-based algorithm. Moreover, we present a network parameter (*i.e.* weights and threshold) initialization scheme for a variety of deep SNN architectures. In the experiment, we show that the proposed initialization scheme appropriately initializes the deep SNNs facilitating training convergence for a given network architecture and training strategy. In addition, as opposed to complex error or neuron normalization method adopted by [116] and [132], respectively, we demonstrate that deep SNNs can be naturally trained by only considering the spiking activities of the network. As a result, our work paves the effective way for training deep SNNs with spike-based BP algorithm.

8.5.2 Spike Activity Analysis

The most important advantage of event-driven operation of neural networks is that the events are very sparse in nature. To verify this claim, we analyzed the spiking activities of the direct-spike trained SNNs and ANN-SNN converted networks in the following subsections.



(a)



(b)

Fig. 8.6.: Layer-wise spike activity in direct-spike trained SNN and ANN-SNN converted network for CIFAR-10 dataset: (a) VGG9 (b) ResNet9 network. The spike activity is normalized with respect to the input layer spike activity which is same for both networks.

Spike Activity per Layer

The layer-wise spike activities of both SNN trained using our proposed methodology and ANN-SNN converted network for VGG9 and ResNet9 are shown in Fig.

8.6a and 8.6b, respectively. In the case of ResNet9, only first average pooling layer's output spike activity is shown in the figure as for the direct-spike trained SNN, the other spatial-poolings are done by stride 2 convolutions. In Fig. 8.6, it can be seen that the input layer has the highest spike activity that is significantly higher than any other layer. The spike activity reduces significantly as the network depth increases.

We can observe from Fig. 8.6a and Fig. 8.6b that the average spike activity in direct-spike trained SNN is much higher than ANN-SNN converted network. The ANN-SNN converted network uses higher threshold compared to 1 (in case of direct-spike trained SNN) since the conversion scheme applies layer-wise neuronal threshold modulation. This higher threshold reduces spike activity in ANN-SNN converted networks. However, in both cases, the spike activity decreases with increasing network depth.

#Spikes/Inference

From Fig. 8.6, it is evident that average spike activity in ANN-SNN converted networks is much less than in SNN trained with our proposed methodology. However, for inference, the network has to be evaluated over a number of time-steps. Therefore, to quantify the actual spike activity for an inference operation, we measured the average number of spikes required for inferring one image. For this purpose, we counted number of spikes generated (including input spikes) for classifying the test set of a particular dataset for a specific number of time-steps and averaged the count for generating the quantity *'#spikes per image inference'*. We have used two different time-steps for ANN-SNN converted VGG networks; one for iso-accuracy comparison and the other one for maximum accuracy comparison with the direct-spike trained SNNs. Iso-accuracy inference requires less #time-steps than maximum accuracy inference, hence has lower number of spikes per image inference. For ResNet networks, the ANN-SNN conversion scheme always provides accuracy less than SNN (trained with proposed algorithm). Hence, we only compare spikes per image inference in

maximum accuracy condition for ANN-SNN converted ResNet networks while comparing with direct-spike trained SNNs. We can quantify the spike-efficiency (amount reduction in #spikes) from the #spikes/image inference. The results are listed in Table 8.8 where, for each network, the 1st row corresponds to iso-accuracy and 2nd row corresponds to maximum-accuracy condition.

Table 8.8.: #Spikes per Image Inference

Dataset	Model	Spike/image		Spike Efficiency
		ANN-SNN	SNN	
MNIST	LeNet	29094	55212	0.53x
		73085		1.32x
SVHN	VGG7	10251782	5564306	1.84x
		16615596		2.99x
	ResNet7	–	4656760	–
		20607244		4.43x
CIFAR-10	VGG9	2226732	1240492	1.80x
		9647563		7.78x
	ResNet9	–	4319988	–
		8745271		2.02x
	ResNet11	–	1531985	–
		8116343		5.30x

Fig. 8.7 shows the relationship between inference accuracy, latency and #spikes/inference for ResNet11 network trained on CIFAR-10 dataset. We can observe that #spikes/inference is higher for direct-spike trained SNN compared to ANN-SNN converted network at any particular latency. However, SNN trained with spike-based BP requires only 100 time-steps for maximum inference accuracy, whereas ANN-SNN converted network requires about 3000 time-steps to reach maximum inference accuracy (which is slightly less than direct-spike trained SNN accuracy). Hence, for maximum-accuracy

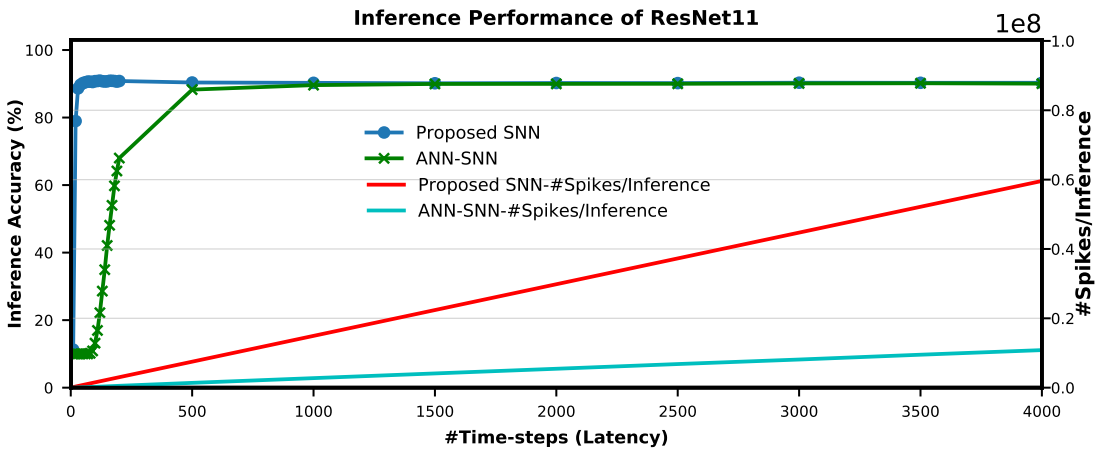


Fig. 8.7.: The comparison of 'accuracy vs latency vs #spikes/inference' for ResNet11 architecture.

condition, direct-spike trained SNN requires much less #spikes/inference compared to ANN-SNN converted network while achieving similar accuracy.

8.5.3 Inference Speedup

The time required for inference is almost linearly proportional to the #time-steps (Fig. 8.7). Hence, we can also quantify the inference speedup for direct-spike trained SNN compared to ANN-SNN converted network from the #time-steps required for inference as shown in Table 8.9. For VGG9 network, we achieve 8x speedup for iso-accuracy and up to 36x speedup in inference for maximum accuracy comparison. Similarly, for ResNet networks we achieve up to 25x-30x speedup in inference.

8.5.4 Complexity Reduction

Deep ANNs struggle to meet the demand of extraordinary computational requirements. SNNs can mitigate this effort by enabling efficient event-driven computations. To compare the computational complexity for these two cases, we first need to under-

Table 8.9.: Inference Speedup

Dataset	Model	Timesteps		Inference
		ANN-SNN	SNN	Speedup
MNIST	LeNet	200	50	4x
		500		10x
SVHN	VGG7	1600	100	16x
		2600		26x
	ResNet7	–	100	–
		2500		25x
CIFAR-10	VGG9	800	100	8x
		3600		36x
	ResNet9	–	100	–
		3000		30x
	ResNet11	–	100	–
		3000		30x

stand the operation principle of both. An ANN operation for inferring the category of a particular input requires a single feed-forward pass per image. For the same task, the network must be evaluated over a number of time-steps in spiking domain. If regular hardware is used for both ANN and SNN, then it is evident that SNN will have computation complexity in the order of hundreds or thousands more compared to an ANN. However, there are specialized hardware that accounts for the event-driven neural operation and 'computes only when required'. SNNs can potentially exploit such alternative mechanisms of network operation and carry out an inference operation in spiking domain much more efficiently than an ANN. Also, for deep SNNs, we have observed the increase in sparsity as the network depth increases. Hence, the benefits from event-driven hardware is expected to increase as the network depth increases.

An estimate of the actual energy consumption of SNNs and comparison with ANNs is outside the scope of this work. However, we can gain some insight by quantifying the energy consumption for a synaptic operation and comparing the number of synaptic operations being performed in the ANN versus the SNN trained with our proposed algorithm and ANN-SNN converted network. We can estimate the number of synaptic operations per layer of a neural network from the structure for the convolutional and linear layers. In an ANN, a multiply-accumulate (MAC) computation is performed per synaptic operation. While, a specialized SNN hardware would perform simply an accumulate computation (AC) per synaptic operation only if an incoming spike is received. Hence, the total number of AC operations in a SNN can be estimated by the layer-wise product and summation of the average neural spike count for a particular layer and the corresponding number of synaptic connections. We also have to multiply the #time-steps with the #AC operations to get total #AC operation for one image inference. Based on this concept, we estimated total number of MAC operations for ANN, and total number of *effective* AC operations for direct-spike trained SNN and ANN-SNN converted network, for VGG9, ResNet9 and ResNet11. The ratio of ANN-SNN converted network AC operations to direct-spike trained SNN AC operations to ANN MAC operations is 28.18:3.61:1 for VGG9 while the ratio is 11.94:5.06:1 for the ResNet9 and 7.26:2.09:1 for ResNet11 (for maximum accuracy condition).

However, a MAC operation usually consumes an order of magnitude more energy than an AC operation. For instance, according to [133], a 32-bit floating point MAC operation consumes 4.6pJ and a 32-bit floating point AC operation consumes 0.9pJ in 45nm technology node. Hence, one synaptic operation in an ANN is equivalent to $\sim$5 synaptic operations in a SNN. Moreover, 32-bit floating point computation can be replaced by fixed point computation using integer MAC and AC units without losing accuracy since the conversion is reported to be almost loss-less [134]. A 32-bit integer MAC consumes roughly 3.2pJ, while a 32-bit AC operation consumes only 0.1pJ in 45nm process technology. Considering this fact, our calculations demon-

strate that the SNNs trained using proposed method will be 7.81x and 8.87x more computationally energy-efficient compared to an ANN-SNN converted network and an ANN, respectively, for the VGG9 network architecture. We also gain 3.47x(2.36x) and 15.32x(6.32x) energy-efficiency, for the ResNet11(ResNet9) network, compared to an ANN-SNN converted network and an ANN, respectively. Fig. 8.8 shows the reduction in computation complexity for ANN-SNN conversion and SNN trained with proposed methodology compared to ANNs.

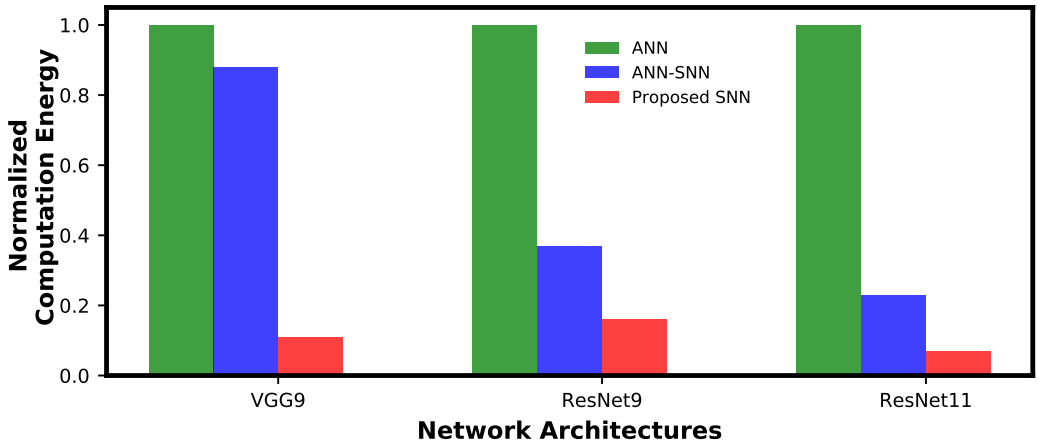


Fig. 8.8.: Inference computation complexity comparison between ANN, ANN-SNN conversion and SNN trained with spike-based backpropagation. ANN computational complexity is considered as baseline for normalization.

It is worth noting here that as the sparsity of the spike signals increases with increase in network depth in SNNs, the energy-efficiency is expected to increase almost exponentially in both ANN-SNN conversion network [102] and SNN trained with proposed methodology compared to an ANN implementation. Hence, depth of network is the key factor for achieving significant increase in the energy efficiency for event-driven SNNs in contrast to ANNs.

Training complexity can also be easily estimated for proposed methodology as it follows the steps in standard ANN backpropagation algorithm. In backpropagation

algorithm, training effort consists of two costs: i) Forward propagation cost and ii) Backward propagation and weight update cost. Using simple cost estimation model based on the number of synaptic operations, we observed that for ANN, backward propagation and weight update cost is ∼2x more expensive compared to forward propagation for a mini-batch size of one. Mini-batch size of one is chosen for simplicity. To get an ANN-SNN converted network, first an ANN is trained using standard backpropagation algorithm and then neuronal threshold modulation is applied to convert it to a SNN. We can neglect the threshold modulation cost as it is a one time cost while training can consist of many epochs. Hence, overall computational complexity of training for ANN-SNN conversion network is similar to ANN. On the other hand, for spike-based backpropagation training scheme, we back-propagate only once in each mini-batch iteration which is computationally same as back-propagation in an ANN. Therefore, the back-propagation and weight update cost for proposed methodology is also same as an ANN. Considering these factors and combining the forward propagation cost (as described earlier), backward propagation and weight update cost, we can estimate the total computational complexity for training. Our estimation shows that, for a mini-batch size of 1, proposed training methodology is ∼1.4x computationally energy efficient compared to an ANN and an ANN-SNN conversion network.

Most state-of-the-art DNN accelerators are focusing on exploiting the computation redundancy in DNN applications to reduce energy consumption. It has been shown that 16 bit inputs, activations and weights achieves similar performance as 32 bit counterparts, while 8 bit inputs, activations and weights are sufficient to achieve optimal quality performance [35]. To that effect, 16 bit and 8 bit multiplications for synaptic operation are replacing the 32 bit multiplication. However, the accumulator width is dependent on the network layer sizes and micro-architecture of the accelerator. Typically, 32 (20) bit accumulator is used for 16 (8) bit multiplication to ensure no overflow in accumulation. Energy numbers for optimized 8 bit and 32 bit fixed point multipliers and accumulators are available in literature for 45 nm process

technology [135]. We have used interpolation to estimate the energy numbers for 20 and 32 bit accumulators and 16 bit multiplier.

Assuming 16 bit multiplier and 32 bit accumulator being used for ANN MAC operation, while 32 bit accumulator being used for SNN AC operation, the ratio of energy consumption per synaptic operation between ANN and SNN will come down to 10.20. Even with this bit precision scaling, the SNNs trained using proposed method achieves up to 4.9x computational energy-efficiency compared to an ANN. On the other hand, assuming 8 bit multiplier and 20 bit accumulator being used for ANN MAC operation, while 20 bit accumulator being used for SNN AC operation, the ratio of energy consumption per synaptic operation between ANN and SNN will come down to 3.87. Our calculations demonstrate that in such case, the SNNs trained using proposed method achieves up to 1.8x computational energy-efficiency compared to an ANN, while converted networks perform worse than ANNs. However, for deeper networks, performance of proposed SNN and converted networks are expected to improve due to the increased sparsity shown in section 8.5.2 and [102].

9. CONCLUSION

9.1 Conclusion and Summary

The aim of this research is to explore energy-efficient hardware and algorithms for Deep Learning. To that effect, we first considered the key limitations in the realization of the existing Deep Learning Networks. The ever-growing complexity of the state-of-the-art deep neural networks (DNNs) together with the explosion in the amount of data to be processed, place significant energy demands on the computing platforms. Therefore, improvement in both training and inference/testing is necessary to exploit the full potential of this emerging paradigm. In the first part of this work, we focused on improving the inference/testing energy-efficiency. In the second part, we aimed at developing training methodologies to facilitate efficient learning.

Approximations at the algorithmic and hardware level can provide energy savings in the inference/testing phase while incurring tolerable accuracy degradation. Retraining the approximate networks, with the approximations in place, helps in mitigating the accuracy loss. We explored algorithmic and hardware level approximations to determine their effectiveness in achieving energy improvements while maintaining the output quality. In particular, we introduced three different approximations, namely, synapse pruning, approximate neuronal multiplication, and voltage-scaled memory for synaptic storage, on DNNs. We also investigated lower complexity networks to explore network approximations. We validated the efficacy of the approximations by comparing the energy benefits with that of optimized DNNs (without approximations). Algorithm (Pruning) and Hardware (Approximate Multiplication, Approximate Memory) level approximations are energy-efficient for high accuracy requirements, while Lower complexity networks are beneficial for low accuracy requirements. Algorithm (Pruning) and Hardware (Approx. Multiplication and

Approx. Memory) level approximations can be combined to get higher energy savings while maintaining reasonable quality. On the contrary, low complexity networks, even though energy efficient, incur severe accuracy loss. Our results clearly indicate that employing properly selected approximations on DNNs leads to improved energy efficiency with competitive classification accuracy.

In recent times, deep learning methods have outperformed traditional machine learning approaches on virtually every single metric. CNNs are one of the chief contributors to this success. To meet the ever-growing demand of solving more challenging tasks, the deep learning networks are becoming larger and larger. However, training of these large networks requires high computational effort and energy requirements. In this work, we exploited the error resiliency of CNN applications and the usefulness of Gabor filters to propose an energy efficient and fast training methodology for CNNs. We designed and implemented several Gabor filter based CNN configurations to obtain the best trade-off between accuracy and energy. We proposed a balanced CNN configuration, where fixed Gabor filters are not only used in the 1st convolutional layer, but also in the latter convolutional layers in conjunction with regular weight kernels. Experiments across various benchmark applications with our proposed scheme demonstrated significant improvements in computational energy consumption during training, and also reduction in training time, storage requirements, and memory access energy for negligible loss in the classification accuracy. Note, since our proposed Gabor kernel based CNN is faster and consumes less energy during training, the cost of retraining the network, when new training data is available, will be less compared to a conventional CNN. Also since fixed kernels are used in 50-66.7% convolution operations, dedicated hardware can be designed to gain more benefits not only in back-propagation, but also during forward-propagation in the network.

Retraining large neural networks to accommodate new, previously unseen data demands high computational time and energy requirements. Also, previously seen training samples may not be available at the time of retraining. In this work, we

also explore an efficient training methodology and incrementally growing a Deep Convolutional Neural Network (DCNN) to allow new tasks to be learned while sharing part of the base network. Our methodology is inspired by transfer learning techniques, although it does not forget previously learned tasks. An updated network for learning new tasks is formed using previously learned convolutional layers (shared from initial part of base network) with addition of few newly added convolutional kernels included in the later layers of the network. Initial experiments show that the classification accuracy achieved on several recognition applications by our approach is comparable to the regular incremental learning approach (where networks are updated with new training samples only, without any network sharing).

While DNNs are usually trained using well established back-propagation algorithm, SNNs are typically trained using STDP based unsupervised process. However, the typical shallow spiking network architectures trained using STDP have limited capacity for expressing complex representations, while training a very deep spiking network has not been successful so far. In this work, we propose a spike-based back-propagation training methodology for state-of-the-art deep SNN architectures. This methodology enables real-time training in deep SNNs while achieving comparable inference accuracies on standard image recognition tasks. Our experiments show the effectiveness of the proposed learning strategy on deeper SNNs by achieving the best classification accuracies in MNIST, SVHN and CIFAR-10 datasets among other networks trained with spike-based learning till date. The performance gap in terms of quality between ANN and SNN is substantially reduced by the application of our proposed methodology. We can achieve *6.32x-15.32x* energy-efficiency compared to ANN counterparts as well as *2.36x-7.81x* over ANN-SNN converted networks for inference by exploiting our training methodology and applying the trained SNN on neuromorphic hardware. Moreover, trained deep SNNs can infer *8x-36x* faster than ANN-SNN converted networks.

9.2 Future Work

Our aim is to develop novel techniques to improve training and inference of Deep NNs further. To that effect, we are proposing several ideas to be explored in the future.

Training deeper networks, with ever-increasing data samples, becoming a big challenge for even state-of-the-art computing platforms. In order to solve complex problems using limited resources (computation power, time, storage capacity etc.), reducing the network size might not be an acceptable solution. However, there is a potential opportunity of reducing the training-effort by intelligently using training samples. May be not all training samples are equally important. Also not all training samples are needed during the whole training process. Banking on these concepts, we will explore sample importance based learning techniques to reduce training effort in Deep Learning.

REFERENCES

REFERENCES

[1] W. S. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity," *The bulletin of mathematical biophysics*, vol. 5, no. 4, pp. 115–133, 1943.

[2] E. Reingold and J. Nightingale, "History of Neural Networks," http://www.psych.utoronto.ca/users/reingold/courses/ai/cache/neural4.html, [Online; accessed 21-April-2019].

[3] K. Strachnyi, "Brief History of Neural Networks," https://medium.com/analytics-vidhya/brief-history-of-neural-networks-44c2bf72eec, 2019, [Online; accessed 21-April-2019].

[4] F. Rosenblatt, "Principles of neurodynamics. perceptrons and the theory of brain mechanisms," Cornell Aeronautical Lab Inc Buffalo NY, Tech. Rep., 1961.

[5] M. Minsky and S. Papert, "An introduction to computational geometry," *Cambridge tiass., HIT*, 1969.

[6] P. Werbos, "Beyond regression:" new tools for prediction and analysis in the behavioral sciences," *Ph. D. dissertation, Harvard University*, 1974.

[7] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning internal representations by error propagation," California Univ San Diego La Jolla Inst for Cognitive Science, Tech. Rep., 1985.

[8] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

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

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

[11] C. Rosenberg, "Improving Photo Search: A Step Across the Semantic Gap," http://googleresearch.blogspot.com/2013/06/improving-photo-search-step-across.html, 2013, [Online; accessed 26-October-2017].

[12] G. Dede and M. H. Sazlı, "Speech recognition with artificial neural networks," *Digital Signal Processing*, vol. 20, no. 3, pp. 763–768, 2010.

[13] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot *et al.*, "Mastering the game of go with deep neural networks and tree search," *Nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[14] J. Misra and I. Saha, "Artificial neural networks in hardware: A survey of two decades of progress," *Neurocomputing*, vol. 74, no. 1, pp. 239–255, 2010.

[15] C. Farabet, B. Martini, B. Corda, P. Akselrod, E. Culurciello, and Y. LeCun, "Neuflow: A runtime reconfigurable dataflow processor for vision," in *Computer Vision and Pattern Recognition Workshops (CVPRW), 2011 IEEE Computer Society Conference on*. IEEE, 2011, pp. 109–116.

[16] R. V. Hoang, D. Tanna, L. C. J. Bray, S. M. Dascalu, and F. C. Harris Jr, "A novel cpu/gpu simulation environment for large-scale biologically realistic neural modeling," *Frontiers in neuroinformatics*, vol. 7, 2013.

[17] K.-H. Kim, S. Gaba, D. Wheeler, J. M. Cruz-Albrecht, T. Hussain, N. Srinivasa, and W. Lu, "A functional hybrid memristor crossbar-array/cmos system for data storage and neuromorphic applications," *Nano letters*, vol. 12, no. 1, pp. 389–395, 2011.

[18] C. D. Wright, P. Hosseini, and J. A. V. Diosdado, "Beyond von-neumann computing with nanoscale phase-change memory devices," *Advanced Functional Materials*, vol. 23, no. 18, pp. 2248–2254, 2013.

[19] S. Venkataramani, S. T. Chakradhar, K. Roy, and A. Raghunathan, "Approximate computing and the quest for computing efficiency," in *Proceedings of the 52nd Annual Design Automation Conference, San Francisco, CA, USA, June 7-11, 2015*, 2015, pp. 120:1–120:6.

[20] J. Park, H. Choo, K. Muhammad, S. Choi, Y. Im, and K. Roy, "Non-adaptive and adaptive filter implementation based on sharing multiplication," in *Acoustics, Speech, and Signal Processing, 2000. ICASSP'00. Proceedings. 2000 IEEE International Conference on*, vol. 1. IEEE, 2000, pp. 460–463.

[21] I. J. Chang, D. Mohapatra, and K. Roy, "A priority-based 6t/8t hybrid sram architecture for aggressive voltage scaling in video applications," *IEEE transactions on circuits and systems for video technology*, vol. 21, no. 2, pp. 101–112, 2011.

[22] V. K. Chippa, D. Mohapatra, A. Raghunathan, K. Roy, and S. T. Chakradhar, "Scalable effort hardware design: Exploiting algorithmic resilience for energy efficiency," in *Proceedings of the 47th Design Automation Conference*. ACM, 2010, pp. 555–560.

[23] V. Gupta, D. Mohapatra, S. P. Park, A. Raghunathan, and K. Roy, "Impact: imprecise adders for low-power approximate computing," in *Proceedings of the 17th IEEE/ACM international symposium on Low-power electronics and design*. IEEE Press, 2011, pp. 409–414.

[24] K. Muhammad, "Algorithmic and architectural techniques for low-power digital signal processing," 1999.

[25] C. H. Sequin and R. D. Clay, "Fault tolerance in artificial neural networks," in *1990 IJCNN International Joint Conference on Neural Networks*, June 1990, pp. 703–708 vol.1.

[26] P. Panda, A. Sengupta, S. S. Sarwar, G. Srinivasan, S. Venkataramani, A. Raghunathan, and K. Roy, "Cross-layer approximations for neuromorphic computing: From devices to circuits and systems," in *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*. IEEE, 2016, pp. 1–6.

[27] S. Venkataramani, A. Ranjan, K. Roy, and A. Raghunathan, "Axnn: energy-efficient neuromorphic systems using approximate computing," in *Proceedings of the 2014 international symposium on Low power electronics and design*. ACM, 2014, pp. 27–32.

[28] S. S. Sarwar, S. Venkataramani, A. Raghunathan, and K. Roy, "Multiplier-less artificial neurons exploiting error resiliency for energy-efficient neural computing," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2016*. IEEE, 2016, pp. 145–150.

[29] S. S. Sarwar, P. Panda, and K. Roy, "Gabor filter assisted energy efficient fast learning convolutional neural networks," in *2017 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*, July 2017, pp. 1–6.

[30] G. Srinivasan, P. Wijesinghe, S. S. Sarwar, A. Jaiswal, and K. Roy, "Significance driven hybrid 8t-6t sram for energy-efficient synaptic storage in artificial neural networks," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2016*. IEEE, 2016, pp. 151–156.

[31] J. Kung, D. Kim, and S. Mukhopadhyay, "A power-aware digital feedforward neural network platform with backpropagation driven approximate synapses," in *Low Power Electronics and Design (ISLPED), 2015 IEEE/ACM International Symposium on*. IEEE, 2015, pp. 85–90.

[32] B. Reagen, P. Whatmough, R. Adolf, S. Rama, H. Lee, S. K. Lee, J. M. Hernández-Lobato, G.-Y. Wei, and D. Brooks, "Minerva: Enabling low-power, highly-accurate deep neural network accelerators," in *Proceedings of the 43rd International Symposium on Computer Architecture*. IEEE Press, 2016, pp. 267–278.

[33] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," *arXiv preprint arXiv:1510.00149*, 2015.

[34] S. S. Sarwar, S. Venkataramani, A. Ankit, A. Raghunathan, and K. Roy, "Energy-efficient neural computing with approximate multipliers," *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, vol. 14, no. 2, p. 16, 2018.

[35] S. S. Sarwar, G. Srinivasan, B. Han, P. Wijesinghe, A. Jaiswal, P. Panda, A. Raghunathan, and K. Roy, "Energy efficient neural computing: A study of cross-layer approximations," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 2018.

[36] S. S. Sarwar, A. Ankit, and K. Roy, "Incremental learning in deep convolutional neural networks using partial network sharing," *arXiv preprint arXiv:1712.02719*, 2017.

[37] C. Lee, S. S. Sarwar, and K. Roy, "Enabling spike-based backpropagation in state-of-the-art deep neural network architectures," *arXiv preprint arXiv:1903.06379*, 2019.

[38] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[39] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[40] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting," *The Journal of Machine Learning Research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[41] S. Song, K. D. Miller, and L. F. Abbott, "Competitive hebbian learning through spike-timing-dependent synaptic plasticity," *Nature neuroscience*, vol. 3, no. 9, p. 919, 2000.

[42] S. Sivanantham, K. Jagannadha Naidu, S. Balamurugan, and D. Bhuvana Phaneendra, "Low power floating point computation sharing multiplier for signal processing applications," *International Journal of Engineering and Technology*, vol. 5, no. 2, pp. 979–85, 2013.

[43] G. Karakonstantis and K. Roy, "An optimal algorithm for low power multiplier-less fir filter design using chebychev criterion," in *Acoustics, Speech and Signal Processing, 2007. ICASSP 2007. IEEE International Conference on*, vol. 2. IEEE, 2007, pp. II–49.

[44] V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines," in *Proceedings of the 27th international conference on machine learning (ICML-10)*, 2010, pp. 807–814.

[45] R. B. Palm, "Prediction as a candidate for learning deep hierarchical models of data," *Technical University of Denmark*, vol. 5, 2012.

[46] A. Vedaldi and K. Lenc, "Matconvnet: Convolutional neural networks for matlab," in *Proceedings of the 23rd ACM international conference on Multimedia*. ACM, 2015, pp. 689–692.

[47] S. R. Nassif, "Modeling and analysis of manufacturing variations," in *Custom Integrated Circuits, 2001, IEEE Conference on.* IEEE, 2001, pp. 223–228.

[48] C. Visweswariah, "Death, taxes and failing chips," in *Proceedings of the 40th annual Design Automation Conference*. ACM, 2003, pp. 343–347.

[49] S. Borkar, T. Karnik, S. Narendra, J. Tschanz, A. Keshavarzi, and V. De, "Parameter variations and impact on circuits and microarchitecture," in *Proceedings of the 40th annual Design Automation Conference*. ACM, 2003, pp. 338–342.

[50] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "Cacti 6.0: A tool to model large caches," *HP Laboratories*, pp. 22–31, 2009.

[51] M. Lin, Q. Chen, and S. Yan, "Network in network," *arXiv preprint arXiv:1312.4400*, 2013.

[52] Z. Lin, M. Courbariaux, R. Memisevic, and Y. Bengio, "Neural networks with few multiplications," *arXiv preprint arXiv:1510.03009*, 2015.

[53] J. Zhang, K. Rangineni, Z. Ghodsi, and S. Garg, "Thundervolt: Enabling aggressive voltage underscaling and timing error resilience for energy efficient deep neural network accelerators," *arXiv preprint arXiv:1802.03806*, 2018.

[54] Y. Lin, S. Zhang, and N. R. Shanbhag, "Variation-tolerant architectures for convolutional neural networks in the near threshold voltage regime," in *Signal Processing Systems (SiPS), 2016 IEEE International Workshop on*. IEEE, 2016, pp. 17–22.

[55] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, "Quantized neural networks: Training neural networks with low precision weights and activations," *arXiv preprint arXiv:1609.07061*, 2016.

[56] Y. Wang, J. Lin, and Z. Wang, "An energy-efficient architecture for binary weight convolutional neural networks," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 26, no. 2, pp. 280–293, Feb 2018.

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

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

[59] T. Liu, S. Fang, Y. Zhao, P. Wang, and J. Zhang, "Implementation of training convolutional neural networks," *arXiv preprint arXiv:1506.01195*, 2015.

[60] G. S. Budhi, R. Adipranata, and F. J. Hartono, "The use of gabor filter and back-propagation neural network for the automobile types recognition," in *2nd International Conference SIIT 2010*, 2010.

[61] B. Kwolek, "Face detection using convolutional neural networks and gabor filters," in *International Conference on Artificial Neural Networks*. Springer, 2005, pp. 551–556.

[62] A. Calderón, S. Roa, and J. Victorino, "Handwritten digit recognition using convolutional neural networks and gabor filters," *Proc. Int. Congr. Comput. Intell*, 2003.

[63] S.-Y. Chang and N. Morgan, "Robust cnn-based speech recognition with gabor filter kernels," in *Fifteenth Annual Conference of the International Speech Communication Association*, 2014.

[64] A. K. Jain, N. K. Ratha, and S. Lakshmanan, "Object detection using gabor filters," *Pattern recognition*, vol. 30, no. 2, pp. 295–309, 1997.

[65] J. G. Daugman, "Uncertainty relation for resolution in space, spatial frequency, and orientation optimized by two-dimensional visual cortical filters," *JOSA A*, vol. 2, no. 7, pp. 1160–1169, 1985.

[66] J. Yosinski, J. Clune, Y. Bengio, and H. Lipson, "How transferable are features in deep neural networks?" in *Advances in neural information processing systems*, 2014, pp. 3320–3328.

[67] M. Haghighat, S. Zonouz, and M. Abdel-Mottaleb, "Identification using encrypted biometrics," in *International Conference on Computer Analysis of Images and Patterns*. Springer, 2013, pp. 440–448.

[68] L. Van der Maaten, "A new benchmark dataset for handwritten character recognition," *Tilburg University*, pp. 2–5, 2009.

[69] A. Krizhevsky and G. Hinton, "Learning multiple layers of features from tiny images," Citeseer, Tech. Rep., 2009.

[70] A. Sharif Razavian, H. Azizpour, J. Sullivan, and S. Carlsson, "Cnn features off-the-shelf: an astounding baseline for recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition workshops*, 2014, pp. 806–813.

[71] S. J. Pan, Q. Yang *et al.*, "A survey on transfer learning," *IEEE Transactions on knowledge and data engineering*, vol. 22, no. 10, pp. 1345–1359, 2010.

[72] N. Patricia and B. Caputo, "Learning to learn, from transfer learning to domain adaptation: A unifying perspective," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2014, pp. 1442–1449.

[73] L. Fei-Fei, R. Fergus, and P. Perona, "One-shot learning of object categories," *IEEE transactions on pattern analysis and machine intelligence*, vol. 28, no. 4, pp. 594–611, 2006.

[74] C. H. Lampert, H. Nickisch, and S. Harmeling, "Learning to detect unseen object classes by between-class attribute transfer," in *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*. IEEE, 2009, pp. 951–958.

[75] R. Polikar, L. Upda, S. S. Upda, and V. Honavar, "Learn++: An incremental learning algorithm for supervised neural networks," *IEEE transactions on systems, man, and cybernetics, part C (applications and reviews)*, vol. 31, no. 4, pp. 497–508, 2001.

[76] D. Medera and S. Babinec, "Incremental learning of convolutional neural networks." in *IJCCI*, 2009, pp. 547–550.

[77] A. Royer and C. H. Lampert, "Classifier adaptation at prediction time," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2015, pp. 1401–1409.

[78] A. Pentina, V. Sharmanska, and C. H. Lampert, "Curriculum learning of multiple tasks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2015, pp. 5492–5500.

[79] T. Xiao, J. Zhang, K. Yang, Y. Peng, and Z. Zhang, "Error-driven incremental learning in deep convolutional neural network for large-scale image classification," in *Proceedings of the 22nd ACM international conference on Multimedia*. ACM, 2014, pp. 177–186.

[80] D. Roy, P. Panda, and K. Roy, "Tree-cnn: A deep convolutional neural network for lifelong learning," *arXiv preprint arXiv:1802.05800*, 2018.

[81] Z. Li and D. Hoiem, "Learning without forgetting," in *European Conference on Computer Vision*. Springer, 2016, pp. 614–629.

[82] S.-A. Rebuffi, A. Kolesnikov, and C. H. Lampert, "icarl: Incremental classifier and representation learning," *arXiv preprint arXiv:1611.07725*, 2016.

[83] K. Shmelkov, C. Schmid, and K. Alahari, "Incremental learning of object detectors without catastrophic forgetting," *arXiv preprint arXiv:1708.06977*, 2017.

[84] A. A. Rusu, N. C. Rabinowitz, G. Desjardins, H. Soyer, J. Kirkpatrick, K. Kavukcuoglu, R. Pascanu, and R. Hadsell, "Progressive neural networks," *arXiv preprint arXiv:1606.04671*, 2016.

[85] J. Kirkpatrick, R. Pascanu, N. Rabinowitz, J. Veness, G. Desjardins, A. A. Rusu, K. Milan, J. Quan, T. Ramalho, A. Grabska-Barwinska *et al.*, "Overcoming catastrophic forgetting in neural networks," *Proceedings of the National Academy of Sciences*, p. 201611835, 2017.

[86] R. Aljundi, P. Chakravarty, and T. Tuytelaars, "Expert gate: Lifelong learning with a network of experts," *arXiv preprint arXiv:1611.06194*, 2016.

[87] P. Panda, J. M. Allred, S. Ramanathan, and K. Roy, "Asp: Learning to forget with adaptive synaptic plasticity in spiking neural networks," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 8, no. 1, pp. 51–64, 2018.

[88] M. Mermillod, A. Bugaiska, and P. Bonin, "The stability-plasticity dilemma: Investigating the continuum from catastrophic forgetting to age-limited learning effects," *Frontiers in psychology*, vol. 4, 2013.

[89] P. Panda, A. Ankit, P. Wijesinghe, and K. Roy, "Falcon: Feature driven selective classification for energy-efficient image recognition," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 36, no. 12, 2017.

[90] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun *et al.*, "Dadiannao: A machine-learning supercomputer," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2014, pp. 609–622.

[91] A. Ankit, I. El Hajj, S. R. Chalamalasetti, G. Ndu, M. Foltin, R. S. Williams, P. Faraboschi, W.-m. Hwu, J. P. Strachan, K. Roy, and D. Milojicic, "PUMA: A programmable ultra-efficient memristor-based accelerator for machine learning inference," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019.

[92] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, "Efficient processing of deep neural networks: A tutorial and survey," *Proceedings of the IEEE*, vol. 105, no. 12, pp. 2295–2329, 2017.

[93] A. Paszke, S. Gross, and S. Chintala, "Pytorch," 2017.

[94] K. He, X. Zhang, S. Ren, and J. Sun, "Identity mappings in deep residual networks," in *European Conference on Computer Vision*. Springer, 2016, pp. 630–645.

[95] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 4700–4708.

[96] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *arXiv preprint arXiv:1704.04861*, 2017.

[97] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," *arXiv preprint arXiv:1502.03167*, 2015.

[98] P. A. Merolla, J. V. Arthur, R. Alvarez-Icaza, A. S. Cassidy, J. Sawada, F. Akopyan, B. L. Jackson, N. Imam, C. Guo, Y. Nakamura *et al.*, "A million spiking-neuron integrated circuit with a scalable communication network and interface," *Science*, vol. 345, no. 6197, pp. 668–673, 2014.

[99] M. Davies, N. Srinivasa, T.-H. Lin, G. Chinya, Y. Cao, S. H. Choday, G. Dimou, P. Joshi, N. Imam, S. Jain *et al.*, "Loihi: A neuromorphic manycore processor with on-chip learning," *IEEE Micro*, vol. 38, no. 1, pp. 82–99, 2018.

[100] W. Maass, "Networks of spiking neurons: the third generation of neural network models," *Neural networks*, vol. 10, no. 9, pp. 1659–1671, 1997.

[101] P. Dayan and L. F. Abbott, *Theoretical neuroscience: computational and mathematical modeling of neural systems*. Cambridge, MA: MIT Press, 2001, vol. 806.

[102] A. Sengupta, Y. Ye, R. Wang, C. Liu, and K. Roy, "Going deeper in spiking neural networks: Vgg and residual architectures," *arXiv preprint arXiv:1802.02627*, 2018.

[103] B. Rueckauer, I.-A. Lungu, Y. Hu, M. Pfeiffer, and S.-C. Liu, "Conversion of continuous-valued deep networks to efficient event-driven networks for image classification," *Frontiers in neuroscience*, vol. 11, p. 682, 2017.

[104] Y. Cao, Y. Chen, and D. Khosla, "Spiking deep convolutional neural networks for energy-efficient object recognition," *International Journal of Computer Vision*, vol. 113, no. 1, pp. 54–66, 2015.

[105] P. U. Diehl, G. Zarrella, A. Cassidy, B. U. Pedroni, and E. Neftci, "Conversion of artificial recurrent neural networks to spiking neural networks for low-power neuromorphic hardware," in *Rebooting Computing (ICRC), IEEE International Conference on*. IEEE, 2016, pp. 1–8.

[106] E. Hunsberger and C. Eliasmith, "Spiking deep networks with lif neurons," *arXiv preprint arXiv:1510.08829*, 2015.

[107] S. B. Furber, D. R. Lester, L. A. Plana, J. D. Garside, E. Painkras, S. Temple, and A. D. Brown, "Overview of the spinnaker system architecture," *IEEE Transactions on Computers*, vol. 62, no. 12, pp. 2454–2467, 2013.

[108] P. U. Diehl and M. Cook, "Unsupervised learning of digit recognition using spike-timing-dependent plasticity," *Frontiers in computational neuroscience*, vol. 9, p. 99, 2015.

[109] B. Zhao, R. Ding, S. Chen, B. Linares-Barranco, and H. Tang, "Feedforward categorization on aer motion events using cortex-like features in a spiking neural network," *IEEE transactions on neural networks and learning systems*, vol. 26, no. 9, pp. 1963–1978, 2015.

[110] J. M. Brader, W. Senn, and S. Fusi, "Learning real-world stimuli in a neural network with spike-driven synaptic dynamics," *Neural computation*, vol. 19, no. 11, pp. 2881–2912, 2007.

[111] G. Srinivasan, P. Panda, and K. Roy, "Spilinc: Spiking liquid-ensemble computing for unsupervised speech and image recognition," *Frontiers in Neuroscience*, vol. 12, p. 524, 2018.

[112] ——, "Stdp-based unsupervised feature learning using convolution-over-time in spiking neural networks for energy-efficient neuromorphic computing," *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, vol. 14, no. 4, p. 44, 2018.

[113] S. R. Kheradpisheh, M. Ganjtabesh, S. J. Thorpe, and T. Masquelier, "Stdp-based spiking deep neural networks for object recognition," *arXiv preprint arXiv:1611.01421*, 2016.

[114] C. Lee, G. Srinivasan, P. Panda, and K. Roy, "Deep spiking convolutional neural network trained with unsupervised spike timing dependent plasticity," *IEEE Transactions on Cognitive and Developmental Systems*, 2018.

[115] S. M. Bohte, J. N. Kok, and H. La Poutre, "Error-backpropagation in temporally encoded networks of spiking neurons," *Neurocomputing*, vol. 48, no. 1-4, pp. 17–37, 2002.

[116] J. H. Lee, T. Delbruck, and M. Pfeiffer, "Training deep spiking neural networks using backpropagation," *Frontiers in neuroscience*, vol. 10, p. 508, 2016.

[117] P. Panda and K. Roy, "Unsupervised regenerative learning of hierarchical features in spiking deep networks for object recognition," in *Neural Networks (IJCNN), 2016 International Joint Conference on*. IEEE, 2016, pp. 299–306.

[118] C. Lee, P. Panda, G. Srinivasan, and K. Roy, "Training deep spiking convolutional neural networks with stdp-based unsupervised pre-training followed by supervised fine-tuning," *Frontiers in Neuroscience*, vol. 12, p. 435, 2018.

[119] Y. Bengio, N. Léonard, and A. Courville, "Estimating or propagating gradients through stochastic neurons for conditional computation," *arXiv preprint arXiv:1308.3432*, 2013.

[120] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," in *Proceedings of the IEEE international conference on computer vision*, 2015, pp. 1026–1034.

[121] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[122] Y. Netzer, T. Wang, A. Coates, A. Bissacco, B. Wu, and A. Y. Ng, "Reading digits in natural images with unsupervised feature learning," in *NIPS workshop on deep learning and unsupervised feature learning*, vol. 2011, no. 2, 2011, p. 5.

[123] G. Orchard, A. Jayawant, G. K. Cohen, and N. Thakor, "Converting static image datasets to spiking neuromorphic datasets using saccades," *Frontiers in neuroscience*, vol. 9, p. 437, 2015.

[124] P. Lichtsteiner, C. Posch, and T. Delbruck, "A 128×128 120 db 15μs latency asynchronous temporal contrast vision sensor," *IEEE journal of solid-state circuits*, vol. 43, no. 2, pp. 566–576, 2008.

[125] Y. Jin, P. Li, and W. Zhang, "Hybrid macro/micro level backpropagation for training deep spiking neural networks," *arXiv preprint arXiv:1805.07866*, 2018.

[126] Y. Wu, L. Deng, G. Li, J. Zhu, and L. Shi, "Spatio-temporal backpropagation for training high-performance spiking neural networks," *Frontiers in neuroscience*, vol. 12, 2018.

[127] E. O. Neftci, C. Augustine, S. Paul, and G. Detorakis, "Event-driven random back-propagation: Enabling neuromorphic deep learning machines," *Frontiers in neuroscience*, vol. 11, p. 324, 2017.

[128] H. Mostafa, "Supervised learning based on temporal coding in spiking neural networks," *IEEE transactions on neural networks and learning systems*, 2017.

[129] A. Tavanaei and A. S. Maida, "Bio-inspired spiking convolutional neural network using layer-wise sparse coding and stdp learning," *arXiv preprint arXiv:1611.03000*, 2016.

[130] ——, "Multi-layer unsupervised learning in a spiking convolutional neural network," in *2017 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 2017, pp. 2023–2030.

[131] S. Esser, P. Merolla, J. Arthur, A. Cassidy, R. Appuswamy, A. Andreopoulos, D. Berg, J. McKinstry, T. Melano, D. Barch *et al.*, "Convolutional networks for fast, energy-efficient neuromorphic computing. 2016," *Preprint on ArXiv. http://arxiv. org/abs/1603.08270. Accessed*, vol. 27, 2016.

[132] Y. Wu, L. Deng, G. Li, J. Zhu, and L. Shi, "Direct training for spiking neural networks: Faster, larger, better," *arXiv preprint arXiv:1809.05793*, 2018.

[133] S. Han, J. Pool, J. Tran, and W. Dally, "Learning both weights and connections for efficient neural network," in *Advances in neural information processing systems*, 2015, pp. 1135–1143.

[134] D. Lin, S. Talathi, and S. Annapureddy, "Fixed point quantization of deep convolutional networks," in *International Conference on Machine Learning*, 2016, pp. 2849–2858.

[135] M. Horowitz, "1.1 computing's energy problem (and what we can do about it)," in *2014 IEEE international solid-state circuits conference digest of technical papers (ISSCC)*. IEEE, 2014, pp. 10–14.

VITA

VITA

Syed Shakib Sarwar Received the B.Sc. and M.Sc. degrees in electrical and electronic engineering from the Bangladesh University of Engineering and Technology (BUET), Dhaka, Bangladesh, in 2012 and 2014, respectively. Since 2014, he has been pursuing a Ph.D. degree in Electrical and Computer Engineering at Purdue University, under the guidance of Prof. Kaushik Roy. In Summer 2018, he worked as research intern in Facebook Reality Labs, where he developed optimization techniques for machine learning applications.

His primary research focus is energy efficient algorithms and hardware implementation for neuromorphic circuits (Deep Learning) based on CMOS and emerging devices. His research interests also include approximate computing in the field of 'Deep Learning'.