EXPLORING METHODS FOR EFFICIENT LEARNING IN NEURAL NETWORKS

by

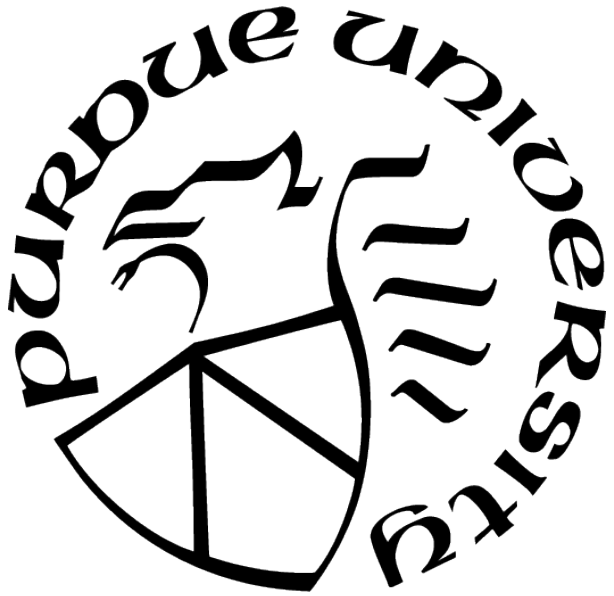
Deboleena Roy

A Dissertation

Submitted to the Faculty of Purdue University

In Partial Fulfillment of the Requirements for the degree of

Doctor of Philosophy



School of Electrical and Computer Engineering

West Lafayette, Indiana

August 2021

THE PURDUE UNIVERSITY GRADUATE SCHOOL
STATEMENT OF COMMITTEE APPROVAL

Dr. Kaushik Roy, Chair

School of Electrical and Computer Engineering

Dr. Anand Raghunathan

School of Electrical and Computer Engineering

Dr. Vijay Raghunathan

School of Electrical and Computer Engineering

Dr. Shreyas Sen

School of Electrical and Computer Engineering

Approved by:

Dr. Dimitrios Peroulis

Dedicated to *Didun* for being my first teacher

ACKNOWLEDGMENTS

First and foremost, I would like to thank Prof. Kaushik Roy for taking me under his guidance in 2016, when I joined Purdue University. Over the years, I have learned several invaluable lessons from him which forged me into a much better researcher. I am forever grateful for the warm welcome I received from his lab, Nanoelectronics Research Laboratory (NRL), and the opportunity to work with a very talented group of people. I would also like to thank my committee members, Prof. Anand Raghunathan, Prof. Vijay Raghunathan, and Prof. Shreyas Sen. I am grateful for the many insightful discussions I had with them which helped steer my Ph.D. research in the right direction. I was also extremely fortunate to have forged lifelong friendships with people I met through this Ph.D. program. I want to thank Abhronil for being my mentor and my friend since my first day. Also, I would like to thank Chamika, Parami, Akhilesh, and Aayush for being the best seniors one could hope for. I deeply cherish all the time we got to spend together. I would also like to thank Indranil, my partner, who I met during my Ph.D. program. He has been my rock, and supported me through all the lows, and celebrated with me all the highs. Finally, I would like to express utmost gratitude towards my parents. They have been my pillars of strength throughout my life, pushing me to chase my dreams.

TABLE OF CONTENTS

Page

Page

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

In the past fifty years, Deep Neural Networks (DNNs) have evolved greatly from a single perceptron to complex multi-layered networks with non-linear activation functions. Today, they form the backbone of Artificial Intelligence, with a diverse application landscape, such as smart assistants, wearables, targeted marketing, autonomous vehicles, etc. The design of DNNs continues to change, as we push its abilities to perform more human-like tasks at an industrial scale.

Multi-task learning and knowledge sharing are essential to human-like learning. Humans progressively acquire knowledge throughout their life, and they do so by remembering, and modifying prior skills for new tasks. In our first work, we investigate the representations learned by Spiking Neural Networks (SNNs), and how to share this knowledge across tasks. Our prior task was MNIST image generation using a spiking autoencoder. We combined the generative half of the autoencoder with a spiking audio-decoder for our new task, i.e audio-to-image conversion of utterances of digits to their corresponding images. We show that objects of different modalities carrying the same meaning can be mapped into a shared latent space comprised of spatio-temporal spike maps, and one can transfer prior skills, in this case, image generation, from one task to another, in a purely Spiking domain. Next, we propose Tree-CNN, an adaptive hierarchical network structure composed of Deep Convolutional Neural Networks(DCNNs) that can grow and learn as new data becomes available. The network organizes the incrementally available data into feature-driven super-classes and improves upon existing hierarchical CNN models by adding the capability of self-growth.

While the above works focused solely on algorithmic design, the underlying hardware determines the efficiency of model implementation. Currently, neural networks are implemented in CMOS based digital hardware such as GPUs and CPUs. However, the saturating scaling trend of CMOS has garnered great interest in Non-Volatile Memory (NVM) technologies such as Spintronics and RRAM. However, most emerging technologies have inherent reliability issues, such as stochasticity and non-linear device characteristics. Inspired by the recent works in spin-based stochastic neurons, we studied the algorithmic impact of designing a neural network using stochastic activations. We trained VGG-like networks on CIFAR-

10/100 with 4 different binary activations and analyzed the trade-off between deterministic and stochastic activations.

NVM-based crossbars further promise fast and energy-efficient in-situ matrix-vector multiplications (MVM). However, the analog nature of computing in these NVM crossbars introduces approximations in the MVM operations, resulting in deviations from ideal output values. We first studied the impact of these non-idealities on the performance of vanilla DNNs under adversarial circumstances, and we observed that the non-ideal behavior interferes with the computation of the exact gradient of the model, which is required for adversarial image generation. In a non-adaptive attack, where the attacker is unaware of the analog hardware, analog computing offered varying degree of intrinsic robustness under all attack scenarios - Transfer, Black Box, and White Box attacks. We also demonstrated "Hardware-in-Loop" adaptive attacks that circumvent this robustness by utilizing the knowledge of the NVM model.

Next, we explored the design of robust DNNs through the amalgamation of adversarial training and the intrinsic robustness offered by NVM crossbar based analog hardware. We studied the noise stability of such networks on unperturbed inputs and observed that internal activations of adversarially trained networks have lower Signal-to-Noise Ratio (SNR), and are sensitive to noise than vanilla networks. As a result, they suffer significantly higher performance degradation due to the non-ideal computations, on an average $2\times$ accuracy drop. On the other hand, for adversarial images, the same networks displayed a $5-10\%$ gain in robust accuracy due to the underlying NVM crossbar when the attack epsilon (ϵ_{attack}, the degree of input perturbations) was greater than the epsilon of the adversarial training (ϵ_{train}). Our results indicate that implementing adversarially trained networks on analog hardware requires careful calibration between hardware non-idealities and ϵ_{train} to achieve optimum robustness and performance.

1. INTRODUCTION

Automation has been the universal the metric of human progress, our progression of tools and machines has been used to define the various periods of human civilization. Today, we are at the precipice of the Fourth Industrial Revolution and Artificial Intelligence (AI) is one of the keys to it [1]. Today AI-driven technology influences many aspects of our life, both personal and communal. Our personal lives are enhanced with smart assistants, wearables, personalized media, self-driving cars, etc, as well as large-scale collective services like AI-powered facial recognition systems for security, predictive trading for markets, targeted advertising, etc. The ubiquity of AI is a recent phenomenon due to major breakthroughs in the sub-field of Neural Networks (NNs) in the past decade[2], [3].

Neural Networks are a sub-class of Machine Learning (ML) algorithms that are inspired from the physiology of neuronal systems in animals. The human brain is the cynosure of AI researchers, as many attempt to understand and replicate its functioning. Starting from the early models, such as Perceptrons, Neural Networks has diverged into two sub-groups, the more popular Deep Learning (DL) or Artificial Neural Networks (ANNs) [4], and the more niche, bio-plausible Spiking Neural Networks (SNNs) [5].

While DNNs have proved themselves as powerful computational tools for many tasks, there is still a significant gap between their capabilities and human-like intelligence. One key distinguishing factor is multi-task learning. Humans learn a wide variety of tasks over time, often utilizing knowledge of the past to adapt to new scenarios. They are capable of remembering the past skills, and simultaneously learning new skills. Neural Networks, on the other hand, are trained on highly specific unique tasks. Adapting to a new task often requires deisgining a new model and training from scratch [6]. In the first half of the thesis, we investigate both ANNs and SNNs with respect to multi-task learning.

Bio-plausibility is the driving force when designing Spiking Neural Networks. The activation function of an SNN is often an Integrate and Fire (IF) neuron, that closely mimics the functionality of a biological neuron. The data within the network is temporal in nature, and represented as a series of spikes, just like the human brain. SNNs offer a promising alternative to current artificial neural networks to enable low-power event-driven neuromor-

phic hardware. Spike-based neuromorphic applications require processing and extracting meaningful information from spatio-temporal data, represented as series of spike trains over time. As our first work, we explore the nature of these hidden representations, and synaptic weights, and if they are transferable across tasks. We stack two networks trained separately on different sub-tasks, and create an audio-to-image synthesis pipeline [7].

Next, We expand this idea of network-sharing to Convolutional Neural Networks (CNNs). We propose Tree-CNN, a hierarchical neural network system composed of multiple CNNs. The CNNs at higher nodes are shared by CNNs of lower nodes, creating a multi-level classification system. This system is useful in incremental learning of new data. The Tree-CNN algorithm identifies a subset of nodes that require retraining to accommodate new information. It limits the changes to a part of the network, and thereby reduces the retraining effort. Thus, in the first half, we proposed techniques that address certain algorithmic limitations in Neural Networks in terms of multi-task and incremental learning. In the second half, we focus on the implementation challenges of current Deep Neural Networks from a hardware perspective. Nowadays many consumer applications rely on deep neural networks (DNNs) to enhance their user experience, such as smart wearables, smart assistants, etc. As our reliance on deep learning increases, so does the need to build secure, reliable, efficient frameworks for executing the intensive computational requirements of DNNs.

This has encouraged researchers to venture beyond traditional von-Neuman computing, and exploit emerging technologies to design specialized hardware for neural networks. In the existing CMOS hardware, computations are digital and accurate in nature. However, the saturating trend of CMOS scaling limits our efforts for more energy efficient CMOS hardware. On the other hand, hardware based on emerging technologies, such as Non-Volatile Memory (NVM) based crossbars [8]–[10], and neuro-mimetic devices [11] suffer from reliability issues. While these physical properties may seem undesirable on the surface, careful algorithm-hardware co-design can help us leverage their stochastic and non-linear behaviour for efficient ad robust implementations of neural networks. To that effect, we investigate deep stochastic neural networks, as they are ideal candidates for implementation on an inherently stochastic hardware [12], [13]. We also analyze the impact of non-idealities

of crossbar based hardware on Neural Network performance, particularly under adversarial attack [14].

The next 5 chapters are organized as follows. In Ch. 2 we present multi-task learning in a spike-based environment using spiking autoencoders. Next, we showcase Tree-CNN, our proposed method of incremental learning using a hierarchical ensemble of Convolutional Neural Networks (CNNs), in Ch. 3. In Ch. 4, we analyze the performance of deep neural networks with stochastic activations. And in Ch. 5 and 6, we look at the relationship between adversarial robustness of DNNs and the non-idealities of analog computing. In the final chapter, Ch. 7, we summarize our key findings and outline the future direction of our research work.

2. TRANSFER LEARNING WITH SPIKING AUTOENCODERS

2.1 Introduction

For any neural network, the first step of learning is the ability to encode the input into meaningful representations. Autoencoders are a class of neural networks that can learn efficient data encodings in an unsupervised manner [15]. Their two-layer structure makes them easy to train as well. Also, multiple autoencoders can be trained separately and then stacked to enhance functionality [16]. In the domain of SNNs as well, autoencoders provide an exciting opportunity for implementing unsupervised feature learning [17]. Hence, we use autoencoders to investigate how input spike trains can be processed and encoded into meaningful hidden representations in a spatio-temporal format of output spike trains which can be used to recognize and regenerate the original input.

Generally, autoencoders are used to learn the hidden representations of data belonging to one modality only. However, the information surrounding us presents itself in multiple modalities - vision, audio, and touch. We learn to associate sounds, visuals and other sensory stimuli to one another. For example, an "apple" when shown as an image, or as text, or heard as an audio, holds the same meaning for us. A better learning system is one that is capable of learning shared representation of multimodal data [18]. [19] proposed a bimodal SNN model that performs person authentication using speech and visual (face) signals. STDP-trained networks on bimodal data have exhibited better performance [20]. In this work, we explore the possibility of two sensory inputs - audio and visual, of the same object, learning a shared representation using multiple autoencoders, and then use this shared representation to synthesize images from audio samples.

To enable the above discussed functionalities, we must look at a way to train these spiking autoencoders. While several prior works exist in training these networks, each comes with its own advantages and drawbacks. One way to train spiking autoencoders is by using Spike Timing Dependent Plasticity (STDP) [21], an unsupervised local learning rule based on spike timings, such as [22] and [23]. However, STDP, being unsupervised and localized, still fails to train SNNs to perform at par with ANNs. Another approach is derived from

ANN backpropagation; the average firing rate of the output neurons is used to compute the global loss [24], [25]. Rate-coded loss fails to include spatio-temporal information of the network, as the network response is accumulated over time to compute the loss. [26] applied backpropagation through time (BPTT) [27], while [28] proposed a hybrid backpropagation technique to incorporate the temporal effects. Very recently [29] demonstrated direct training of deep SNNs in a Pytorch based implementation framework. However, it continues to be a challenge to accurately map the time-dependent neuronal behavior with a time-averaged rate coded loss function.

In a network trained for classification, an output layer neuron competes with its neighbors for the highest firing rate, which translates into the class label, thus making rate-coded loss a requirement. However, the target for an autoencoder is very different. The output neurons are trained to regenerate the input neuron patterns. Hence, they provide us with an interesting opportunity where one can choose not to use rate-coded loss. Spiking neurons have an internal state, referred to as the membrane potential (V_{mem}), that regulates the firing rate of the neuron. The V_{mem} changes over time depending on the input to the neuron, and whenever it exceeds a threshold, the neuron generates a spike. [17] first presented a backpropagation algorithm for spiking autoencoders that uses V_{mem} of the output neurons to compute the loss of the network. They proposed an approximate gradient descent based algorithm to learn hierarchical representations in stacked convolutional autoencoders. For training the autoencoders in this work, we compute the loss of the network using V_{mem} of the output neurons, and we incorporate BPTT [27] by unrolling the network over time to compute the gradients.

In this work [7], we demonstrate that in a spike-based environment, inputs can be transformed into compressed spatio-temporal spike maps, which can be then be utilized to reconstruct the input later, or can be transferred across network models, and data modalities. We train and test spiking autoencoders on MNIST and Fashion-MNIST dataset. We also present an audio-to-image synthesis framework, composed of multi-layered fully-connected spiking neural networks. A spiking autoencoder is used to generate compressed spatio-temporal spike maps of images (MNIST). A spiking audiocoder then learns to map audio samples to these compressed spike map representations, which are then converted back to images with

high fidelity using the spiking autoencoder. To the best of our knowledge, this is the first work to perform audio to image synthesis in a spike-based environment.

The chapter is organized in the following manner: In Sec. 2.2, the neuron model, the network structure and notations are introduced. The backpropagation algorithm is explained in detail. This is followed by Sec. 2.3 where the performance of these spiking autoencoders is evaluated on MNIST [30] and Fashion-MNIST [31] datasets. We then setup our Audio to Image synthesis model and evaluate it for converting TI-46 digits audio samples to MNIST images. Finally, in Sec. 2.4, we conclude with discussion on this work and its future prospects.

2.2 Learning Spatio-Temporal Representations using Spiking Autoencoders

In this section, we understand the spiking dynamics of the autoencoder network and mathematically derive the proposed training algorithm, a membrane-potential based backpropagation.

2.2.1 Input Encoding and Neuron Model

A spiking neural network differs from a conventional ANN in two main aspects - inputs and activation functions. For an image classification task, for example, an ANN would typically take the raw pixel values as input. However, in SNNs, inputs are binary spike events that happen over time. There are several methods for input encoding in SNNs currently in use, such as rate encoding, rank order coding and temporal coding [32]. One of the most common methods is rate encoding, where each pixel is mapped to a neuron that produces a Poisson spike train, and its firing rate is proportional to the pixel value. In this work, every pixel value of $0-255$ is scaled to a value between $[0,1]$ and a corresponding Poisson spike train of fixed duration, with a pre-set maximum firing rate, is generated (Fig.2.1). The neuron model is that of a leaky integrate-and-fire (LIF) neuron. The membrane potential (V_{mem}) is the internal state of the neuron that gets updated at each time step based on the input of the neuron, $Z^{[t]}$ (eq. 2.1).The output activation ($A^{[t]}$) of the neuron depends on whether V_{mem} reaches a threshold (V_{th}) or not. At any time instant, the output of the

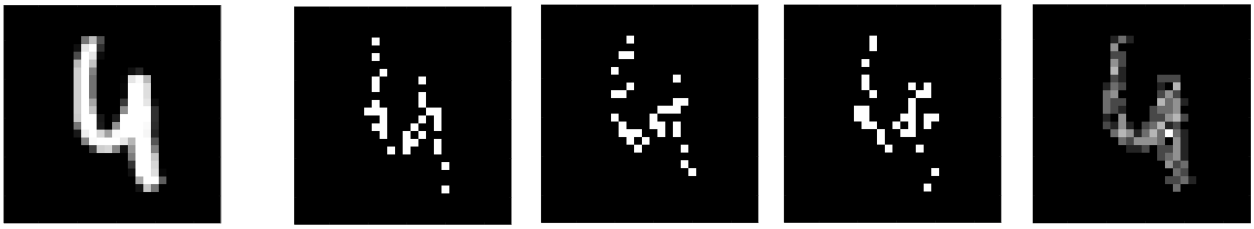


Figure 2.1. : The input image is converted into a spike map over time. At each time step neurons spike with a probability proportional to the corresponding pixel value at their location. These spike maps, when summed over several time steps, reconstruct the original input

neuron is 0 unless the following condition is fulfilled, $V_{mem} \geq V_{th}$ (eq. 2.2). The leak factor is determined by a constant α. After a neuron spikes, it's membrane potential is reset to 0. Fig. 2.2B illustrates a typical neuron's behavior over time.

$$V_{mem}^{[t]} = (1 - \alpha)V_{mem}^{[t-1]} + Z^{[t]} \tag{2.1}$$

$$A^{[t]} = \begin{cases} 0, & V_{mem}^{[t]} < V_{th} \\ 1, & V_{mem}^{[t]} \geq V_{th} \end{cases} \tag{2.2}$$

The activation function (eq. 2.2), which is a clip function, is non-differentiable with respect to V_{mem}, and hence we cannot take its derivative during backpropagation. Several works use various approximate pseudo-derivatives, such as piece-wise linear [33], and exponential derivative [34]. As mentioned in [34], the probability density function of the switching activity of the neuron with respect to its membrane potential can be used to approximate the clip function. It has been observed that biological neurons are noisy and exhibit a probabilistic switching behaviour [35], [36], which can be modeled as having a sigmoid-like characterstic [13]. Thus, for backpropagation, we approximate the clip function (eq. 2.2) with a sigmoid

which is centered around V_{th}, and thereby, the derivative of $A^{[t]}$ is approximated as the derivative of the sigmoid, ($A^{[t]}_{apx}$) (eq. 2.3, 2.4).

$$A^{[t]}_{apx} = \frac{1}{1 + \exp(-(V^{[t]}_{mem} - V_{th}))} \tag{2.3}$$

$$\frac{\partial A^{[t]}}{\partial V^{[t]}_{mem}} \approx \frac{\partial A^{[t]}_{apx}}{\partial V^{[t]}_{mem}} = \frac{\exp(-(V^{[t]}_{mem} - V_{th}))}{(1 + \exp(-(V^{[t]}_{mem} - V_{th})))^2} \tag{2.4}$$

2.2.2 Network Model

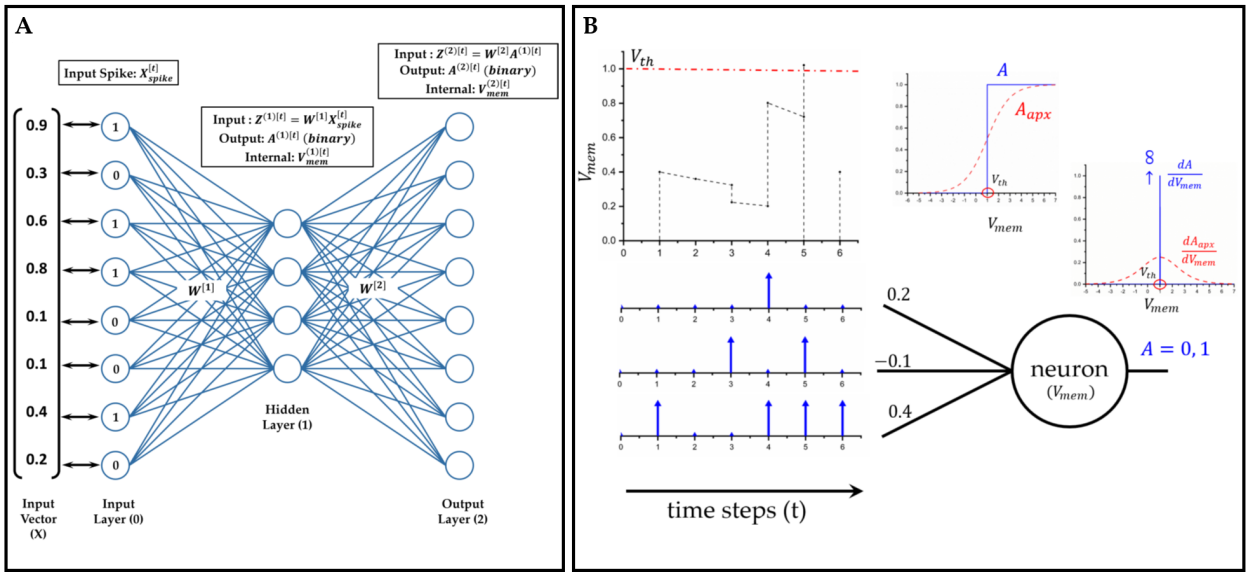


Figure 2.2. : The dynamics of a spiking neural network (SNN): **(A)** A two layer feed-forward SNN at any given arbitrary time instant. The input vector is mapped one-to-one to the input neurons ($layer^{(0)}$). The input value governs the firing rate of the neuron, i.e. number of times the neuron output is 1 in a given duration. **(B)** A leaky integrate and fire (LIF) neuron model with 3 synapses/weights at its input. The membrane potential of the neuron integrates over time (with leak). As soon as it crosses V_{th}, the neuron output changes to 1, and V_{mem} is reset to 0. For taking derivative during backpropagation, a sigmoid approximation is used for the neuron activation

We define the autoencoder as a two layer fully connected feed-forward network. To evaluate our proposed training algorithm, we have used two datasets - MNIST [30] and Fashion MNIST [31]. The two datasets have the same input size, a 28×28 gray-scale

image. Hence, the input and the output layers of their networks have 784 neurons each. The number of $layer^{(1)}$ neurons is different for the two datasets. The input neurons ($layer^{(0)}$) are mapped to the image pixels in a one-to-one manner and generate the Poisson spike trains. The autoencoder trained on MNIST later used as one of the building blocks of the audio-to-image synthesis network. The description of the network and the notation used throughout this chapter is given in Fig. 2.2A.

2.2.3 Backpropagation using Membrane Potential

In this work, loss is computed using the membrane potential of output neurons at every time step and then it's gradient with respect to weights is backpropagated for weight update. The input image is provided to the network as 784×1 binary vector over T time steps, represented as $X^{(t)}_{spike}$. At each time step the desired membrane potential of the output layer is calculated (eq. 2.5). The loss is the difference between the desired membrane potential and the actual membrane potential of the output neurons. Additionally a masking function is used that helps us focus on specific neurons at a time. The mask used here is bitwise XOR between expected spikes ($X^{[t]}_{spike}$) and output spikes ($A^{(2)[t]}$) at a given time instant. The mask only preserves the error of those neurons that either were supposed to spike but did not spike, or were not supposed to spike, but spiked. It sets the loss to be zero for all other neurons. We observed that masking is essential for training in spiking autoencoder as shown in Fig. 2.4A

$$O^{[t]} = V_{th}.^{*}X^{[t]}_{spike} \tag{2.5}$$

$$mask = bitXOR(X^{[t]}_{spike}, A^{(2)[t]}) \tag{2.6}$$

$$Error = E = mask.^{*}(O^{[t]} - V^{(2)[t]}_{mem}) \tag{2.7}$$

$$Loss = L = \frac{1}{2}|E|^2 \tag{2.8}$$

The weight gradients, $\frac{\partial L}{\partial W}$, are computed by back-propagating loss in the two layer network as depicted in Fig. 2.2A. We derive the weight gradients below.

$$\frac{\partial L}{\partial V_{mem}^{(2)[t]}} = -E \tag{2.9}$$

From eq. 2.1,

$$\frac{\partial V_{mem}^{(2)[t]}}{\partial W^{(2)}} = (1-\alpha)\frac{\partial V_{mem}^{(2)[t-1]}}{\partial W^{(2)}} + \left[A^{(1)[t]}\right]^T. \tag{2.10}$$

The derivative is dependent not only on the current input ($A^{(1)[t]}$), but also on the state from previous time step ($V_{mem}^{(2)[t-1]}$).

Next we apply chain rule on eq. 2.9 - 2.10,

$$\frac{\partial L}{\partial W^{(2)}} = \frac{\partial L}{\partial V_{mem}^{(2)[t]}}\frac{\partial V_{mem}^{(2)[t]}}{\partial W^{(2)}} = -E\left[(1-\alpha)\frac{\partial V_{mem}^{(2)[t-1]}}{\partial W^{(2)}} + \left[A^{(1)[t]}\right]^T\right], \tag{2.11}$$

from eq. 2.1,

$$\frac{\partial V_{mem}^{(2)[t]}}{\partial Z^{(2)[t]}} = I, \tag{2.12}$$

from 2.9 and 2.12, we obtain the local error of $layer^{(2)}$ with respect to the overall loss which is backpropagated to $layer^{(1)}$,

$$\delta_2 = \frac{\partial L}{\partial Z^{(2)[t]}} = I(-E) = -E, \tag{2.13}$$

next, the gradients for $layer^{(1)}$ are calculated,

$$\frac{\partial Z^{(2)[t]}}{\partial A^{(1)[t]}} = W^{(2)}, \tag{2.14}$$

from eq. 2.3 - 2.4,

$$\frac{\partial A^{(1)[t]}}{\partial V_{mem}^{(1)[t]}} \approx \frac{\partial A_{apx}^{(1)[t]}}{\partial V_{mem}^{(1)[t]}} = \frac{\exp(-(V_{mem}^{(1)[t]} - V_{th}))}{(1+\exp(-(V_{mem}^{(1)[t]} - V_{th})))^2}, \tag{2.15}$$

from eq. 2.1,

$$\frac{\partial V_{mem}^{(1)[t]}}{\partial W^{(1)}} = (1-\alpha)\frac{\partial V_{mem}^{(1)[t-1]}}{\partial W^{(1)}} + \left[X_{spike}^{[t]}\right]^T, \tag{2.16}$$

from 2.13 - 2.16,

$$\frac{\partial L}{\partial W^{(1)}} = \frac{\partial L}{\partial V_{mem}^{(1)[t]}} \frac{\partial V_{mem}^{(1)[t]}}{\partial W^{(1)}} = \left[\left[W^{(2)} \right]^T \delta_2 \circ \frac{\partial A^{(1)[t]}}{\partial V_{mem}^{(1)[t]}} \right] \left[(1-\alpha) \frac{\partial V_{mem}^{(1)[t-1]}}{\partial W^{(1)}} + \left[X_{spike}^{[t]} \right]^T \right]. \quad (2.17)$$

Thus, equations 2.11 and 2.17 show how gradients of the loss function with respect to weights are calculated. For weight update, we use mini-batch gradient descent and a weight decay value of 1e-5. We implement Adam optimization [37], but the first and second moments of the weight gradients are averaged over time steps per batch (and not averaged over batches). We store $\frac{\partial V_{mem}^{(l)[t]}}{\partial W^{(l)}}$ of the current time step for use in next time step. The initial condition is, $\frac{\partial V_{mem}^{(l)[0]}}{\partial W^{(l)}} = 0$. If a neuron spikes, it's membrane potential is reset and therefore we reset $\frac{\partial V_{mem}^{(l,m)[t]}}{\partial W^{(l)}}$ to 0 as well, where l is the layer number and m is the neuron number.

2.3 Experiments

2.3.1 Regenerative Learning with Spiking Autoencoders

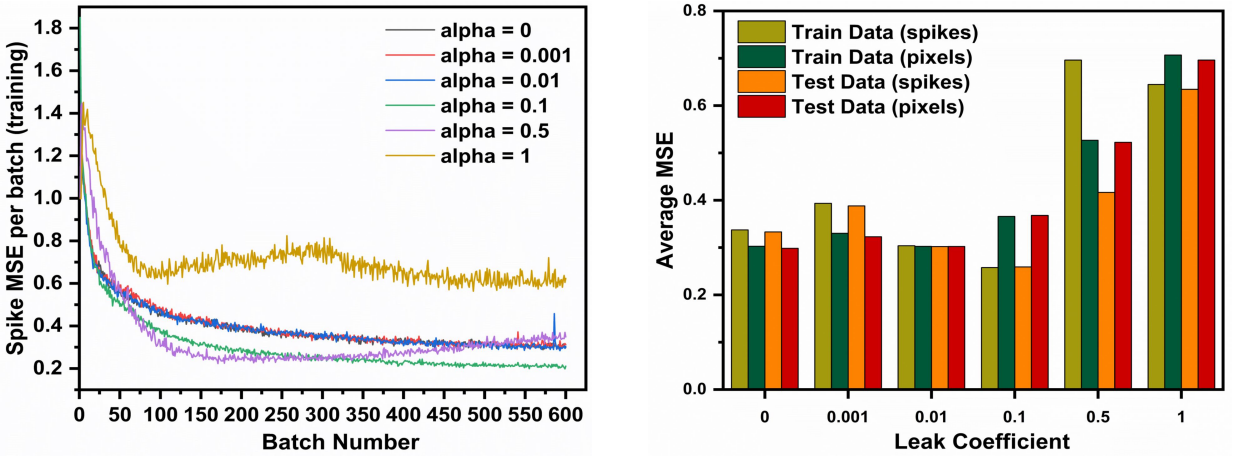


Figure 2.3. : The AE-SNN (784-196-784) is trained over MNIST (60,000 training samples, batch size = 100) for different leak coefficients (α). **(A)** spike-based MSE (Mean Square Error) Reconstruction Loss per batch during training. **(B)** Average MSE over entire dataset after training

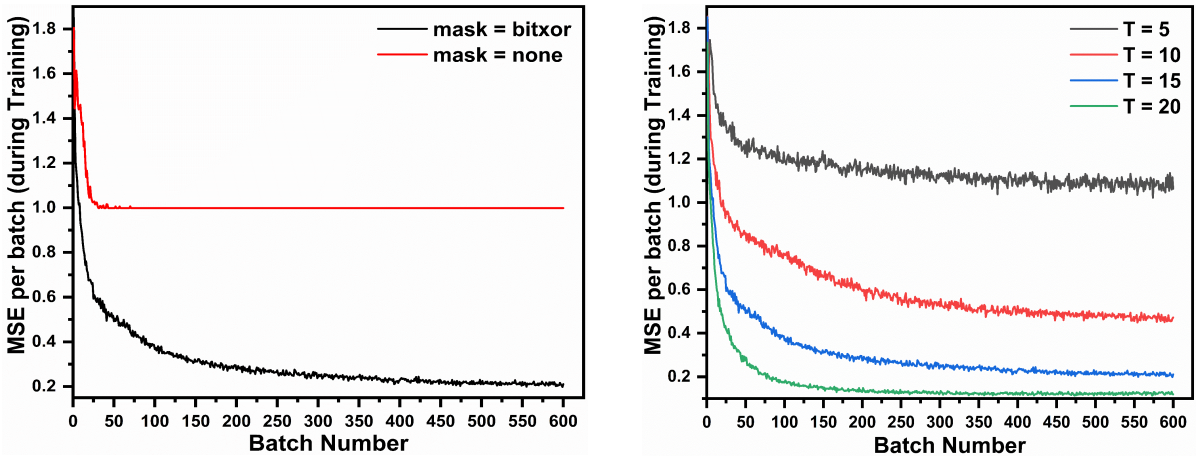


Figure 2.4. : The AE-SNN (784-196-784) is trained over MNIST (60,000 training samples, batch size = 100) and we study the impact of **(A)** mask, and **(B)** input spike train duration on the Mean Square Error (MSE) Reconstruction Loss

For MNIST, a 784-196-784 fully connected network is used. The spiking autoencoder (AE-SNN) is trained for 1 epoch with a batch size of 100, learning rate 5e-4, and a weight decay of 1e-4. The threshold (V_{th}) is set to 1. We define two metrics for network performance, Spike-MSE and MSE. Spike-MSE is the mean square error between the input spike map and the output spike map, both summed over the entire duration. MSE is the mean square error between the input image and output spike map summed over the entire duration. Both, input image and output map, are normalized, zero mean and unit variance, and then the mean square error is computed. The duration of inference is kept the same as the training duration of the network.

It is observed in Fig. 2.3 that the leak coefficient plays an important role in the performance of the network. While a small leak coefficient improves performance, too high of a leak degrades it greatly. We use Spike-MSE as the comparison metric during training in Fig. 2.3A, to observe how well the autoencoder can recreate the input spike train. In Fig. 2.3B, we report two different MSEs, one computed against input spike map (spikes) and the other compared firing rate to pixel values (pixels), after normalizing both. For 'IF' neuron ($\alpha = 0$), the train data performs worse than test data, implying underfitting. At α set to

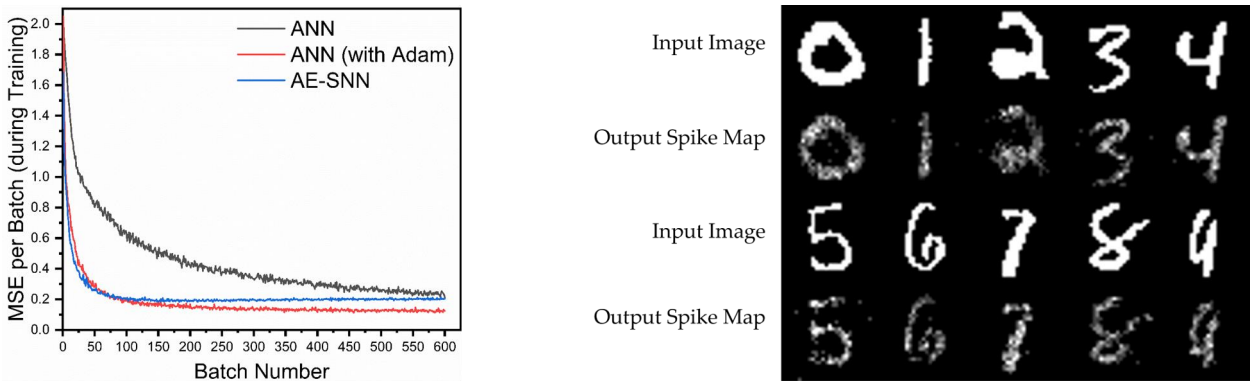


Figure 2.5. : AE-SNN trained on MNIST (training examples = 60,000, batch size = 100). **(A)** Spiking autoencoder (AE-SNN) versus AE-ANNs (trained with/without Adam). **(B)** Regenerated images from test set for AE-SNN (input spike duration = 15, leak = 0.1)

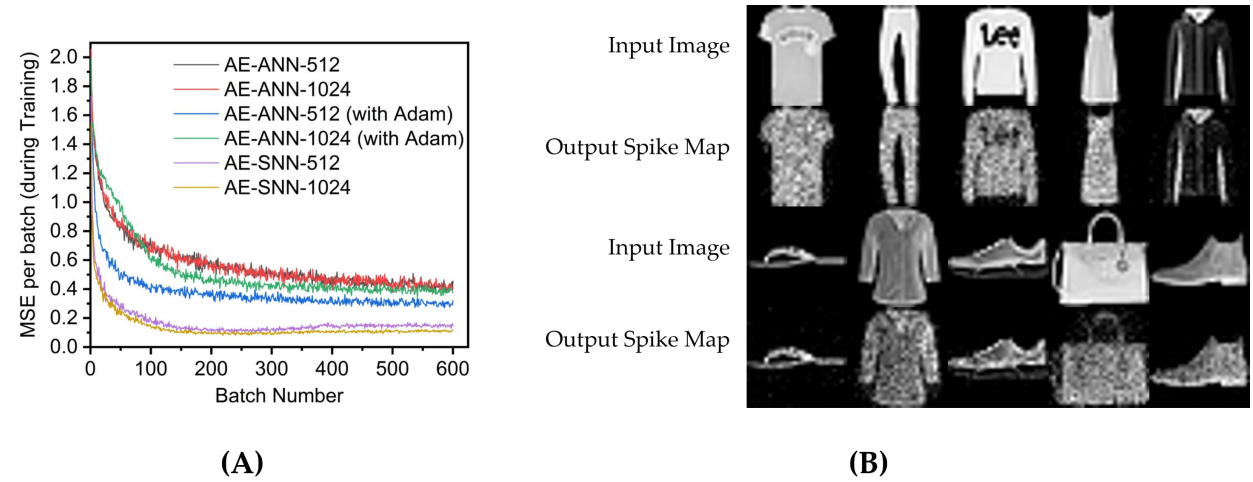


Figure 2.6. : AE-SNN trained on Fashion-MNIST (training examples = 60,000, batch size = 100) **(A)** AE-SNN (784×(512/1024)×784) versus AE-ANNs (trained with/without Adam, lr = 5e-3) **(B)** Regenerated images from test set for AE-SNN-1024

0.01 we find the network having comparable performance between test and train datasets, indicating a good fit. At $\alpha = 0.1$, the Spike-MSE is lowest for both test and train data, however the MSE is higher. While the network is able to faithfully reconstruct the input spike pattern, the difference between Spike-MSE and regular MSE is because of the difference in actual pixel intensity and the converted spike maps generated by the poisson generator at the input. On further increasing the leak, there is an overall performance degradation

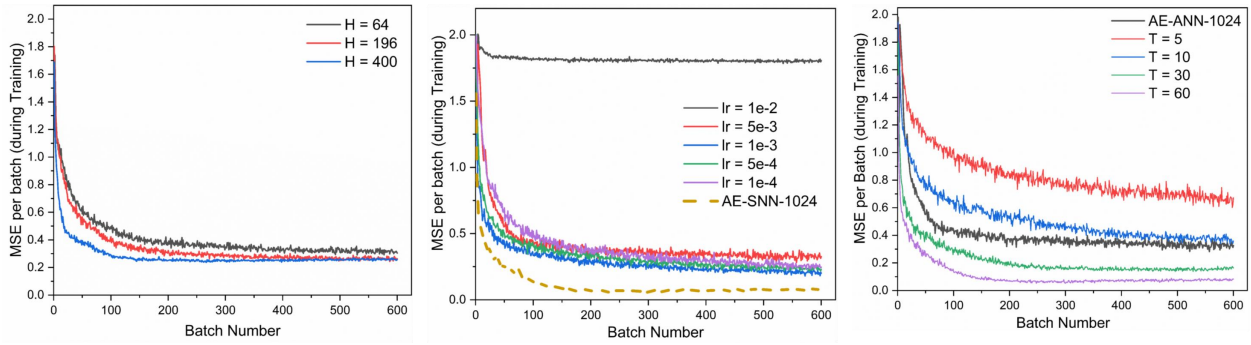


Figure 2.7. : **(A)** AE-SNN (784×H×784) trained on MNIST (training examples = 60,000, batch size = 100) for different hidden layer sizes = 64, 196, 400 **(B)** AE-ANN (784×1024×784) trained on Fashion-MNIST (training examples = 60,000, batch size = 100) with Adam optimization for various learning rates (lr). Baseline: AE-SNN trained with input spike train duration of 60 time steps. **(C)** AE-SNN (784×1024×784) trained on Fashion-MNIST (training examples = 60,000, batch size = 100) for varying input time steps, T = 15, 30, 60. Baseline: AE-ANN trained using Adam with lr = 5e-3

on both test and train data. Thus, we observe that leak coefficient needs to be fine-tuned for optimal performance. Going forth, we set the leak coefficient at 0.1 for all subsequent simulations, as it gave the lowest train and test data MSE on direct comparison with input spike maps.

Fig. 2.4A shows that using a mask function is essential for training this type of network. Without a masking function the training loss does not converge. This is because all of the 784 output neurons are being forced to have membrane potential of 0 or V_{th}, resulting in a highly constrained optimization space, and the network eventually fails to learn any meaningful representations. In the absence of any masking function, the sparsity of the error vector E was less than 5%, whereas, with the mask, the average sparsity was close to 85%. This allows the optimizer to train the critical neurons and synapses of the network. The weight update mechanism learns to focus on correcting the neurons that do not fire correctly, which effectively reduces the number of learning variables, and results in better optimization.

Another interesting observation was that increasing the duration of the input spike train improves the performance as shown in Fig.2.4B. However, it comes at the cost of increased

training time as backpropagation is done at each time step, as well as increased inference time. We settle for an input time duration of 15 as a trade-off between MSE and time taken to train and infer for the next set of simulations.

We also study the impact of hidden layer size for the reconstruction properties of the autoencoder. As shown in Fig. 2.7A, as we increase the size of the network, the performance improves. However, this comes at the cost of increased network size, longer training time and slower inference. While one gets a good improvement when increasing hidden layer size from 64 to 196, the benefit diminishes as we increase the hidden layer size to 400 neurons. Thus for our comparison with ANNs, we use the 784×196×784 network.

For comparison with ANNs, a network (AE-ANN) of same size (784×196×784) is trained with SGD, both with and without Adam optimizer [37] on MNIST for 1 epoch with a learning rate of 0.1, batch size of 100, and weight decay of 1e-4. When training the AE-SNN, the first and second moments of the gradients are computed over sequential time steps within a batch (and not across batches). Thus it is not analogous to the AE-ANN trained with Adam, where the moments are computed over batches. Hence, we compare our network with both variants of the AE-ANNs, trained with and without Adam. The AE-SNN achieves better performance than the AE-ANN trained without Adam; however it lags behind the AE-ANN optimized with Adam as shown in Fig. 2.5A. Some of the reconstructed MNIST images are depicted in Fig. 2.5B. One important thing to note is that the AE-SNN is trained at every time step, hence there are 15× more backpropagation steps as compared to an AE-ANN. However at every backpropagation step, the AE-SNN only backpropagates the error vector of a single spike map, which is very sparse, and carries less information than the error vector of the AE-ANN.

Next, the spiking autoencoder is evaluated on the Fashion-MNIST dataset [31]. It is similar to MNIST, and comprises of 28×28 gray-scale images (60,000 training, 10,000 testing) of clothing items belonging to 10 distinct classes. We test our algorithm on two network sizes: 784-512-784 (AE-SNN-512) and 784-1024-784 (AE-SNN-1024). The AE-SNNs are compared against AE-ANNs of the same sizes (AE-ANN-512, AE-ANN-1024) in Fig. 2.6A. For the AE-SNNs, the duration of input spike train is 60, leak coefficient is 0.1, and learning rate is set at 5e-4. The networks are trained for 1 epoch, with a batch size of 100. The

longer the spike duration, the better would be the spike image resolution. For a duration of 60 time steps, a neuron can spike anywhere between zero to 60 times, thus allowing 61 gray-scale levels. Some of the generated images by AE-SNN-1024 are displayed in Fig. 2.6B. The AE-ANNs are trained for 1 epoch, batch size 100, learning rate 5e-3 and weight decay 1e-4.

For Fashion-MNIST, the AE-SNNs exhibited better performance than AE-ANNs as shown in Fig. 2.6A. We varied the learning rate for AE-ANN, and the AE-SNN still outperformed it's ANN counterpart (Fig. 2.7B). This is an interesting observation, where the better performance comes at the increased effort of per-batch training. Also it exhibits such behavior on only this dataset, and not on MNIST (Fig.2.5A). The spatio-temporal nature of training over each time step could possibly train the network to learn the details in an image better. Spiking Neural Networks have an inherent sparsity in them which could possibly acts like a dropout regularizer [38]. Also, in case of AE-SNN, the update is made at every time step (60 updates per batch), in contrast to ANN where there is one update for one batch. We evaluated AE-SNN for shorter time steps, and observe that for smaller time steps (T = 5, 10), AE-SNN performs worse than AE-ANN (Fig. 2.7C). The impact of time steps is greater for Fashion-MNIST, as compared to MNIST (Fig. 2.4B), as Fashion-MNIST data has more grayscale levels than the near-binary MNIST data. We also observed that, for both datasets, MNIST and Fashion-MNIST, the AE-SNN converges faster than AE-ANNs trained without Adam, and converges at almost the same time as an AE-ANN trained with Adam. The proposed spike-based backpropagation algorithm is able to bring the AE-SNN performance at par, and at times even better, than AE-ANNs.

2.3.2 Audio to Image Synthesis using Spiking Auto-Encoders

Dataset

For the audio to image conversion task, we use two standard datasets, the 0-9 digits subset of TI-46 speech corpus [39] for audio samples, and MNIST dataset [30] for images. The audio dataset has read utterances of 16 speakers for the 10 digits, with a total 4136 audio samples. We divide the audio samples into 3500 train samples and 636 test samples,

maintaining an 85%/15% train/test ratio. For training, we pair each audio sample with an image. We chose two ways of preparing these pairs, as described below:

1. **Dataset A**: 10 unique images of the 10 digits is manually selected (1 image per class) and audio samples are paired with the image belonging to their respective classes (one-image-per-audio-class). All audio samples of a class are paired with the identical image of a digit belonging to that class.
2. **Dataset B**: Each audio sample of the training set is paired with a randomly selected image (of the same label) from the MNIST dataset (one-image-per-audio-sample). Every audio sample is paired with a unique image of the same class.

The testing set is same for both Dataset A and B, comprising of 636 audio samples. All the audio clips were preprocessed using Auditory Toolbox [40]. They were converted to spectrograms having 39 frequency channels over 1500 time steps. The spectrogram is then converted into a 58500×1 vector of length 58500. This vector is then mapped to the input neurons ($layer^{(0)}$) of the audiocoder, which then generate Poisson spike trains over the given training interval.

Network Model

The principle of stacked autoencoders is used to perform audio-to-image synthesis. An autoencoder is built of two sets of weights; the $layer^{(1)}$ weights ($W^{(1)}$) encodes the information into a "hidden state" of a different dimension, and the second layer ($W^{(2)}$) decodes it back to it's original representation. We first train a spiking autoencoder on MNIST dataset. We use the AE-SNN as trained in Fig. 2.5A. Using $layer^{(1)}$ weights ($W^{[1]}$) of this AE-SNN, we generate "hidden-state" representations of the images belonging to the training set of the multimodal dataset. These hidden-state representations are spike trains of a fixed duration. Then we construct an audiocoder: a two layer spiking network that converts spectrograms to this hidden state representation. The audiocoder is trained with membrane potential based backpropagation as described in Sec. 2.2.3. The generated representation, when fed to the "decoder" part of the autoencoder, gives us the corresponding image. The network model is illustrated in Fig. 2.8

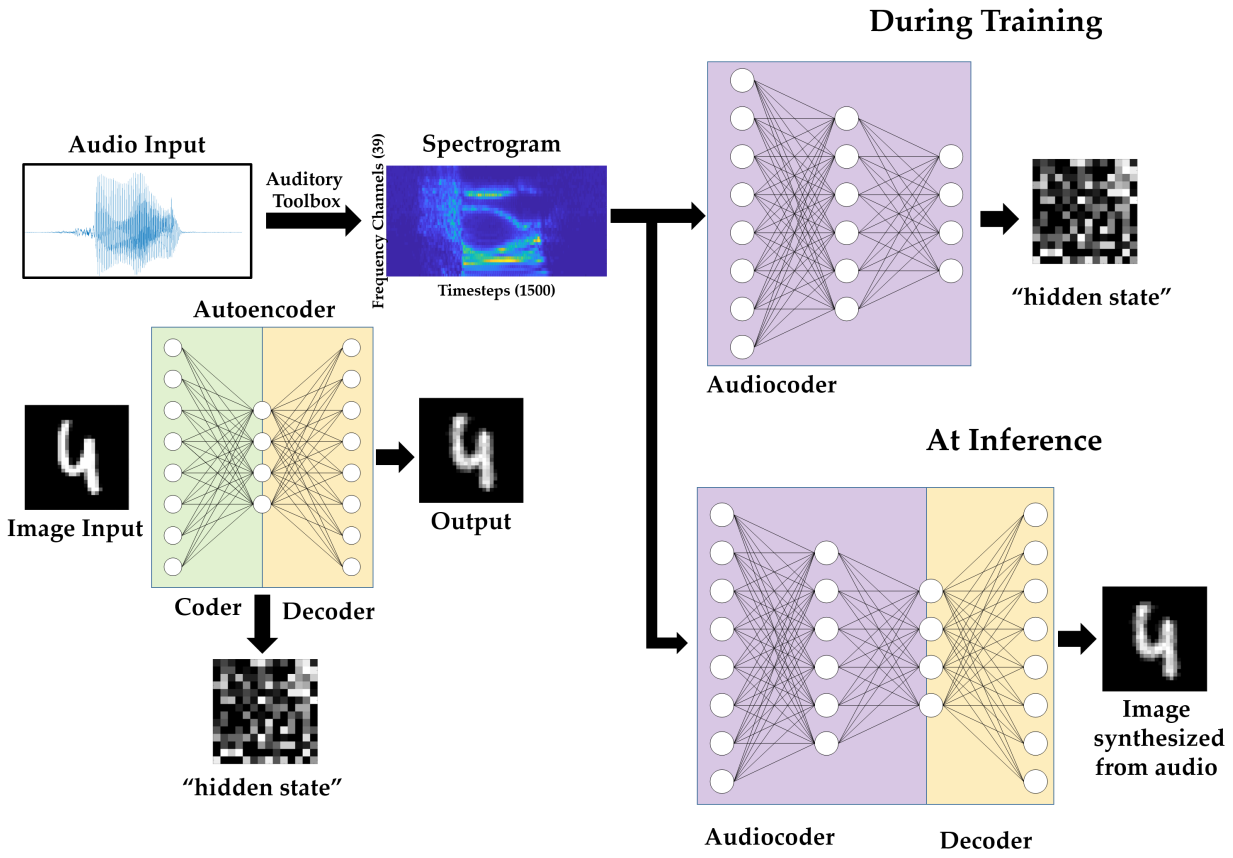


Figure 2.8. : Audio to Image synthesis model using an Autoencoder trained on MNIST images, and an Audiocoder trained to convert TI-46 digits audio samples into corresponding hidden state of the MNIST images.

Results

The MNIST autoencoder (AE-SNN) used for audio-to-image synthesis task is trained using the following parameters: batch size of 100, learning rate 5e-4, leak coefficient 0.1, weight decay 1e-4, input spike train duration 15, and number of epochs 1, as used in Sec. 2.3.1. We use Dataset A and Dataset B (as described in Sec. 2.3.2) to train and evaluate our audio-to-image synthesis model. The images that were paired with the training audio samples are converted to Poisson spike trains (duration 15 time steps) and fed to the AE-SNN, which generates a 196×15 corresponding bitmap as the output of $layer^{(1)}$ (Fig. 2.2A). This spatio temporal representation is then stored. Instead of storing the entire duration of

15 time steps, one can choose to store a subset, such as first 5 or 10 time steps. We use T_h to denote the saved hidden state's duration.

This stored spike map serves as the target spike map for training the audiocoder (AC-SNN), which is a 58500×2048×196 fully connected network. The spectrogram (39×1500) of each audio sample was converted to 58500×1 vector which is mapped one-to-one to the input neurons($layer^{(0)}$). These input neurons then generate Poisson spike trains for 60 time steps. The target map, of T_h time steps, was shown repeatedly over this duration. The audiocoder (AC-SNN) is trained over 20 epochs, with a learning rate of 5e-5 and a leak coefficient of 0.1. Weight decay is set at 1e-4 and the batch size is 50. Once trained, the audiocoder is then merged with $W^{(2)}$ of AE-SNN to create the audio-to-image synthesis model (Fig. 2.8).

For Dataset A, we compare the images generated by audio samples of a class against the MNIST image of that class to compute the MSE. In case of Dataset B, each audio sample of the train set is paired with an unique image. For calculating training set MSE, we compare the paired image and the generated image. For testing set, the generated image of an audio sample is compared with all the training images having the same label in the dataset, and the lowest MSE is recorded. The output spike map is normalized and compared with the normalized MNIST images, as was done previously. Our model gives lower MSE for Dataset A compared to Dataset B (Fig 2.9A), as it is easier to learn just one representative image for a class, than unique images for every audio sample. The network trained with Dataset A generates very good identical images for audio samples belonging to a class. In comparison the network trained on Dataset B generates a blurry image, thus indicating that it has learned to associate the underlying shape and structure of the digits, but has not been able to learn finer details better. This is because the network is trained over multiple different images of the same class, and it learns what is common among them all. Fig. 2.9B displays the generated output spike map for the two models trained over Dataset A and B for 50 different test audio samples (5 of each class).

The duration (T_h) of stored "hidden state" spike train was varied from 15 to 10, 5, 2, and 1. A spike map at a single time step is a 1-bit representation. The AE-SNN compresses an 784×8 bit representation into 196×T_h-bit representation. For T_h = 15, 10, 5, 2, and 1, the compression is 2.1×, 3.2×, 6.4×, 16× and 32× respectively. In Fig. 2.10A we observe

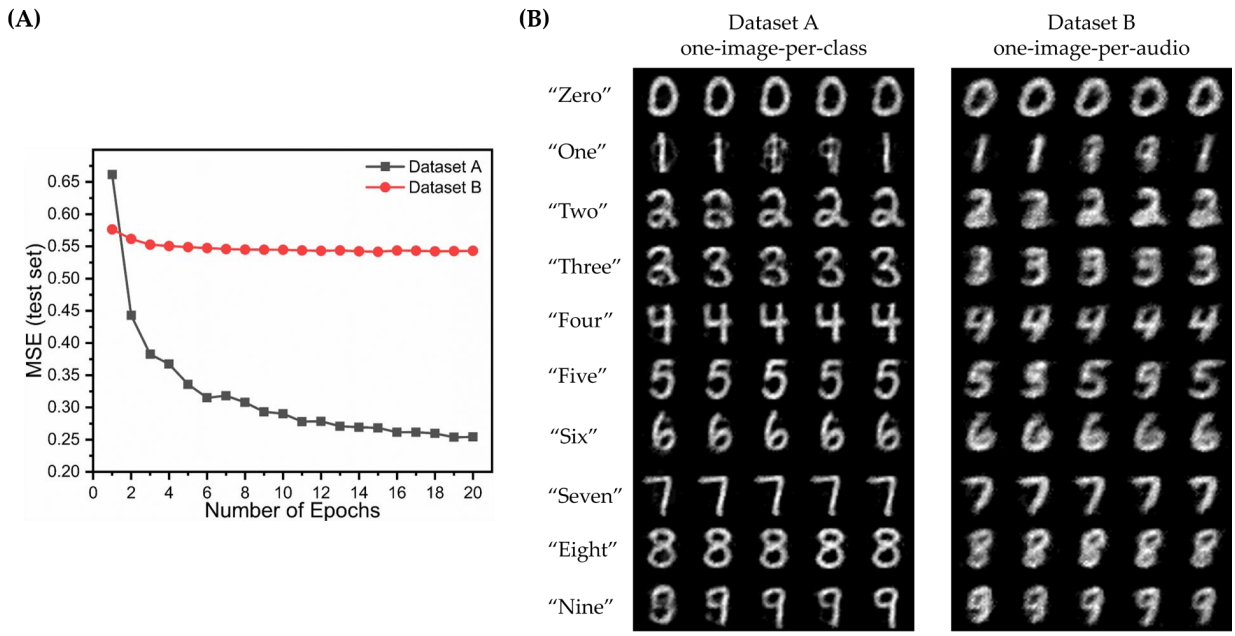


Figure 2.9. : The performance of the Audio to Image synthesis model on the two datasets - A and B ($T_h = 10$)) **(A)** Mean square error loss (test set) **(B)** Images synthesized from different test audio samples (5 per class) for the two datasets A, and B

the reconstruction loss (test set) over epochs for training using different lengths of hidden state. Even when the AC-SNN is trained with a much smaller "hidden state", the AE-SNN is able to reconstruct the images without much loss.

For comparison, we initialize an ANN audiocoder (AC-ANN) of size 58500×2048×196. The AE-ANN trained over MNIST in Sec. 2.3.1 is used to convert the images of the multimodal dataset (A/B) to 196×1 "hidden state" vectors. Each element of this vector is 16 bit full precision number. In case of AE-SNN, the "hidden state" is represented as a $196 \times T_h$ bit map. For comparison, we quantize the equivalent hidden state vector into 2^{T_h} levels. The AC-ANN is trained using these quantized hidden state representations with the following learning parameters: learning rate 1e-4, weight decay 1e-4, batch size 50, epochs 20. Once trained, the ANN audio-to-image synthesis model is built by combining AC-ANN and $layer^{(2)}$ weights ($W^{(2)}$) of AE-ANN. The AC-ANN is trained with/without Adam optimizer, and is paired with the AE-ANN trained with/without Adam optimizer respectively. In Fig. 2.10B, we see that our spiking model achieves a performance in between the two ANN models, a trend we have observed earlier while training autoencoders on MNIST. In

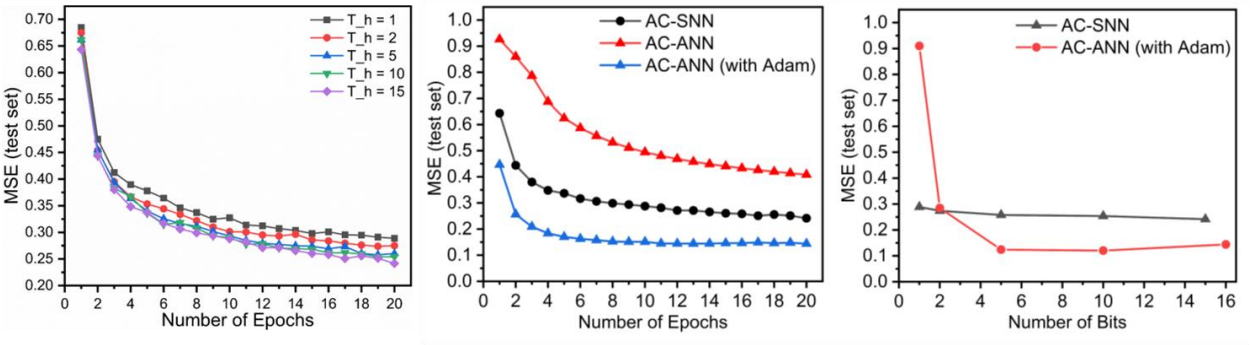


Figure 2.10. : The audiocoder (AC-SNN/AC-ANN) is trained over Dataset A, while the autoencoder (AE-SNN/AE-ANN) is fixed. MSE is reported on the overall audio-to-image synthesis model composed of AC-SNN/ANN and AE-SNN/ANN. **(A)** Reconstruction loss of the audio-to-image synthesis model for varying T_h **(B)** Audiocoder performance AC-SNN ($T_h = 15$) vs AC-ANN (16 bit full precision) **(C)** Effect of training with reduced hidden state representation on AC-SNN and AC-ANN models

this case, the AC-SNN is trained with T_h as 15, while AC-ANNs are trained without any output quantization; both are trained on Dataset A. In Fig. 2.10C, we observe the impact of quantization for the ANN model and the corresponding impact of lower T_h for SNN. For higher hidden state bit precision, the ANN model outperforms the SNN one. However for extreme quantization case, number of bits = 2, and 1, the SNN performs better. This could possibly be attributed to the temporal nature of SNN, where the computation is event-driven and spread out over several time steps.

Note, all simulations were performed using MATLAB, which is a high level simulation environment. The algorithm, however, is agnostic of implementation environment from a functional point of view and can be easily ported to more traditional ML frameworks such as PyTorch or TensorFlow.

2.4 Conclusion

In this work, we propose a method to synthesize images in spike-based environment. In Table 2.1, we have summarized the results of training autoencoders and audiocoders

Table 2.1. : Summary of results obtained for the 3 tasks - Autoencoder on MNIST, Autoencoder on Fashion-MNIST, and Audio to Image conversion (T = input duration for SNN)

				Loss (MSE) (test)		
Dataset	Network Size	Epochs	Timesteps	SNN	ANN	ANN (with Adam)
MNIST	784-196-784	1	15	0.357	0.226	**0.122**
Fashion-MNIST	784-512-784	1	60	**0.178**	0.416	0.300
Fashion-MNIST	784-1024-784	1	60	**0.140**	0.418	0.387
Audio-to-Image A	58500-2048-196/196-784	20	30	0.254	0.408	**0.144**
Audio-to-Image B	58500-2048-196/196-784	20	30	**0.543**	0.611	0.556

using our own V_{mem}-based backpropagation method[1][2]. The proposed algorithm brings SNN performance at par with ANNs for the given tasks, thus depicting the effectiveness of the training algorithm. We demonstrate that spiking autoencoders can be used to generate reduced-duration spike maps ("hidden state") of an input spike train, which are a highly compressed version of the input, and they can be utilized across applications. This is also the first work to demonstrate audio to image synthesis in spiking domain. While training these autoencoders, we made a few important and interesting observations; the first one is the importance of bit masking of the output layer. Trying to steer the membrane potentials of all the neurons is extremely hard to optimize, and selectively correcting only incorrectly spiked neurons makes training easier. This could be applicable to any spiking neural network with a large output layer. Second, while the AE-SNN is trained with spike durations of 15 time steps, we can use hidden state representations of much lower duration to train our audiocoder with negligible loss in reconstruction of images for the audio-to-image synthesis task. In this task, the ANN model trained with Adam outperformed the SNN one when trained with full precision "hidden state". However, at ultra-low precision, the hidden state loses it's meaning in ANN domain, but in SNN domain, the network can still learn from it. This observation raises important questions on the ability of SNNs to possibly compute with less data. While sparsity during inference has always been an important aspect of SNNs, this work suggests that sparsity during training can also be potentially exploited by SNNs.

[1]↑Table 1: Audio-to-Image A: SNN: $T_h = 15$, ANN : no quantization for hidden state
[2]↑Table 1: Audio-to-Image B: SNN: $T_h = 10$, ANN : no quantization for hidden state

We explored how SNNs can be used to compress information into compact spatio-temporal representations and then reconstruct that information back from it. Another interesting observation is that we can potentially train autoencoders and stack them to create deeper spiking networks with greater functionalities. This could be an alternative approach to training deep spiking networks. Thus, this work sheds light on the interesting behavior of spiking neural networks, their ability to generate compact spatio-temporal representations of data, and offers a new training paradigm for learning meaningful representations of complex data.

3. TREE-CNN: A HIERARCHICAL DEEP CONVOLUTIONAL NEURAL NETWORK FOR INCREMENTAL LEARNING

3.1 Introduction

Today, with increased access to large amounts of labeled data (eg. ImageNet [41] contains 1.2 million images with 1000 categories), supervised learning has become the leading paradigm in training DCNNs for image recognition. Traditionally, a DCNN is trained on a dataset containing a large number of labeled images. The network learns to extract relevant features and classify these images. This trained model is then used on real world unlabeled images to classify them. In such training, all the training data is presented to the network during the same training process. However, in real world, we hardly have all the information at once, and data is, instead, gathered incrementally over time. This creates the need for models that can learn new information as it becomes available. In this work, we try to address the challenge of learning on such incrementally available data in the domain of image recognition using deep networks.

A DCNN embeds feature extraction and classification in one coherent architecture within the same model. Modifying one part of the parameter space immediately affects the model globally [42]. Another problem of incrementally training a DCNN is the issue of "catastrophic forgetting" [6]. When a trained DCNN is retrained exclusively over new data, it results in the destruction of existing features learned from earlier data. This mandates using previous data when retraining on new data.

To avoid catastrophic forgetting, and to leverage the features learned in previous task, this work [43] proposes a network made of CNNs that grows hierarchically as new classes are introduced. The network adds the new classes like new leaves to the hierarchical structure. The branching is based on the similarity of features between new and old classes. The initial nodes of the *Tree-CNN* assign the input into coarse super-classes, and as we approach the leaves of the network, finer classification is done. Such a model allows us to leverage the convolution layers learned previously to be used in the new bigger network.

The rest of the chapter is organized as follows. The related work on incremental learning in deep neural networks is discussed in Sec. 3.2. In Sec. 3.3 we present our proposed

network architecture and incremental learning method. In Sec. 3.4, the two experiments using CIFAR-10 and CIFAR-100 datasets are described. It is followed by a detailed analysis of the performance of the network and its comparison with transfer learning and fine tuning in Sec. 3.5. Finally, Sec. 3.6 discusses the merits and limitations of our network, our findings, and possible opportunities for future work.

3.2 Related Work

The modern world of digitized data produces new information every second [44], thus fueling the need for systems that can learn as new data arrives. Traditional deep neural networks are static in that respect, and several new approaches to incremental learning are currently being explored. "One-shot learning" [45] is a Bayesian transfer learning technique, that uses very few training samples to learn new classes. Fast R-CNN [46], a popular framework for object detection, also suffers from "catastrophic forgetting". One way to mitigate this issue is to use a frozen copy of the original network compute and balance the loss when new classes are introduced in the network [47]. "Learning without Forgetting" [48] is another method that uses only new task data to train the network while preserving the original capabilities. The original network is trained on an extensive dataset, such as ImageNet [41], and the new task data is a much smaller dataset. "Expert Gate" [49] adds networks (or experts) trained on new tasks sequentially to the system and uses a set of gating autoencoders to select the right network ("expert") for the given input. Progressive Neural Networks [50] learn to solve complex sequences of task by leveraging prior knowledge with lateral connections. Another recent work on incremental learning in neural networks is "iCaRL" [51], where they built an incremental classifier that can potentially learn incrementally over an indefinitely long time period.

It has been observed that initial layers of a CNN learn very generic features [52] that has been exploited for transfer learning [53], [54]. Common features, that are shared between images, have been used previously to build hierarchical classifiers. These features can be grouped semantically, such as in [55], or be feature-driven, such as "FALCON" [56]. Similar to the progression of complexity of convolutional layers in a DCNN, the upper nodes of a hier-

archical CNN classify the images into coarse super-classes using basic features, like grouping green-colored objects together, or humans faces together. Then deeper nodes perform finer discrimination, such as "boy" v/s "girl" , "apples" v/s "oranges", etc. Such hierarchical CNN models have been shown to perform at par or even better than standard DCNNs [57]. "Discriminative Transfer Learning" [58] is one of the earliest works where classes are categorized hierarchically to improve network performance. Deep Neural Decision Forests [59] unified decision trees and deep CNN's to build a hierarchical classifier. "HD-CNN" [57], is a hierarchical CNN model that is built by exploiting the common feature sharing aspect of images. However, in these works, the dataset is fixed from the beginning, and prior knowledge of all the classes and their properties is used to build a hierarchical model.

In our work, *Tree-CNN* starts out as a single root node and generates new hierarchies to accommodate the new classes. Images belonging to the older dataset are required during retraining, but by localizing the change to a small section of the whole network, our method tries to reduce the training effort and complexity. In [42], a similar approach is applied, where the new classes are added to the old classes, and divided into two super-classes, by using an error-based model. The initial network is cloned to form two new networks which are fine tuned over the two new super-classes. While their motivation was a "divide-and-conquer" approach for large datasets, our work tries to incrementally grow with new data over multiple learning stages. In the next section, we lay out in detail our design principle, network topology and the algorithm used to grow the network.

3.3 Incremental Learning Model

3.3.1 Network Architecture

Inspired from hierarchical classifiers, our proposed model, *Tree-CNN* is composed of multiple nodes connected in a tree-like manner. Each node (except leaf nodes) has a DCNN which is trained to classify the input to the node into one of it's children. The root node is the highest node of the tree, where the first classification happens. The image is then passed on to its child node, as per the classification label. This node further classifies the image, until we reach a leaf node, the last step of classification. Branch nodes are intermediary

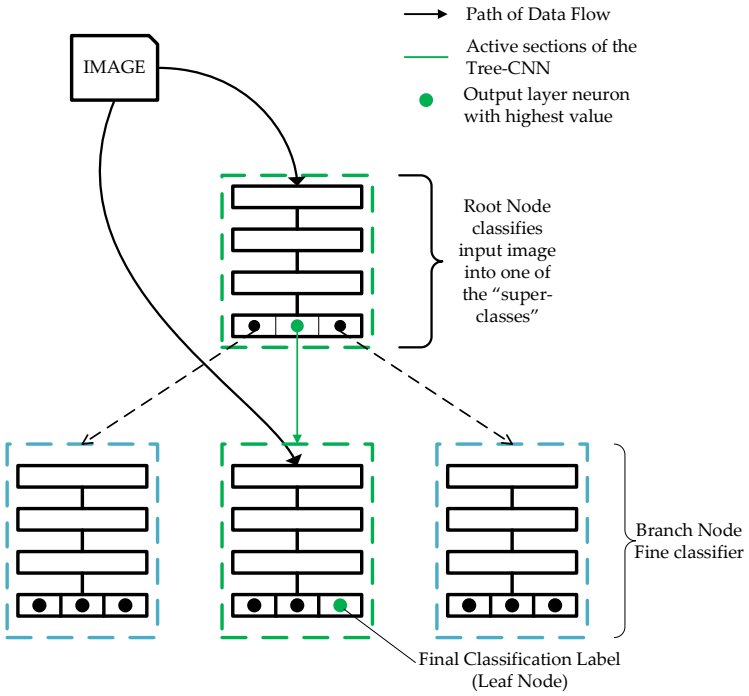


Figure 3.1. : A generic model of 2-level Tree-CNN: The output of the root node is used to select the branch node at the next level.

nodes, each having a parent and two or more children. The leaf node is the last level of the tree. Each leaf node is uniquely associated to a class and no two leaf nodes have the same class. Fig. 3.1 shows the root node and branch nodes for a two-stage classification network. Each output of the second level branch node is a leaf node, which is the output node of the branch CNN. The inference methodology of such a network is given by Algorithm 1.

3.3.2 The Learning Algorithm

We start with the assumption that we have a model that is already trained to recognize a certain number of objects. The model could be hierarchical with multiple CNNs or could be just a single CNN acting as a root node with multiple leaf nodes. A new task is defined as learning to identify images belonging to M new classes. We start at the root node of our given model, and we provide a small sample of images ($\sim 10\%$) from the new training set as input to this node.

Algorithm 1 Tree-CNN: At Inference

I = Input Image, $node$ = Root Node of the Tree
procedure ClassPredict($I, node$)
    $count$ = # of children of node
    if $count = 0$ **then**
        $label$ = class label of the node
        return $label$
    else
        $nextNode = EvaluateNode(I, node)$
         returns the address of the child node of highest output neuron
        return $ClassPredict(I, nextNode)$
    end if
end procedure

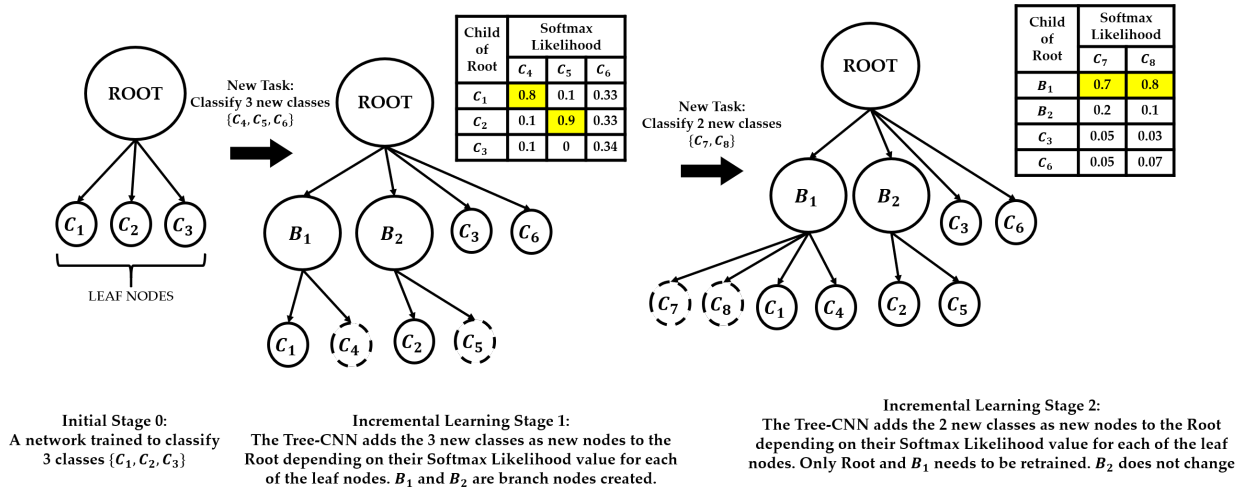


Figure 3.2. : An example illustrating multiple incremental learning stages of the Tree-CNN. The network starts as a single root node, and expands as new classes are added.

We obtain a 3 dimensional matrix from the output layer, $O^{K\times M\times I}$, where, K is the number of children of the root node, M is the number of new classes, and I is the number of sample images per class. $O(k, m, i)$ denotes the output of the k^{th} neuron for the i^{th} image belonging to the m^{th} class where $k \in [1, K]$, $m \in [1, M]$, and $i \in [1, I]$. $O_{avg}^{K\times M}$ is the average of the outputs over I images. Softmax likelihood is computed over O_{avg} (eq. 3.1) to obtain the likelihood matrix $L^{K\times M}$ (eq. 3.2).

$$O_{avg}(k, m) = \sum_{i=1}^{I} \frac{O(k, m, i)}{I} \tag{3.1}$$

$$L(k, m) = \frac{e^{O_{avg}(k,m)}}{\sum_{k=1}^{K} e^{O_{avg}(k,m)}} \tag{3.2}$$

$$\tag{3.3}$$

We generate an ordered list S from $L^{K\times M}$, having the following properties

- The list S has M objects. Each object corresponds uniquely to one of the new M classes.
- Each object $S[i]$ has the following attributes:
  - $S[i].label$ = label of the new class

  - $S[i].value = [v_1, v_2, v_3]$, top 3 average softmax (o_{avg}) output values for that class in descending order, $v_1 \geq v_2 \geq v_3$
  - $S[i].nodes = [n_1, n_2, n_3]$, output nodes corresponding to the softmax outputs v_1, v_2, v_3
- S is ordered in the decreasing value of $S[i].value[1]$

The ordering is done to ensure that new classes with high likelihood values are added first to *Tree-CNN*. Softmax likelihood is used instead of number of images that get classified as each of the child nodes because it translates the output layer's response to the images into an exponential scale and helps us better identify how similar an image is to one of the already existing labels. After constructing S, we look at its first element, $S[1]$, and take one of the 3 actions.

i. **Add the new class to an existing child node:** If v_1 is greater than the next value (v_2) by a threshold, α (a design specification), that class indicates a strong resemblance/association with a particular child node. The new class is added to corresponding child node n_1.

ii. **Merge two child nodes to form a new child node and add the new class to this node:** If there are more than 1 child nodes that the new class has a strong likelihood for, we can combine them to form a new child node. It happens when $v_1 - v_2 < \alpha$, and $v_2 - v_3 > \beta$ (another threshold, defined by the user). For example, if the top 3 likelihood values were $v_1 = 0.48$, $v_2 = 0.45$, and $v_3 = 0.05$. Then, provided n_2 is a leaf node, we merge n_2 into n_1, and add the new class to n_1.

iii. **Add the new class as a new child node:** If the new class doesn't have a likelihood value that is greater than other values by a good margin ($v_1 - v_2 < \alpha, v_2 - v_3 < \beta$), or all child nodes are full, the network expands horizontally by adding the new class as a new child node. This node will be a leaf node.

As the root node keeps adding new branches and sub-branches, the branch nodes with more children tend to get heavier. Incoming new classes tend to have a higher softmax likelihood for branch nodes with greater number of children. To prevent the *Tree-CNN* from becoming lop-sided, one can set the maximum number of children a branch node can have.

When calculating $L(k, m)$, we substitute $e^{O_{avg}(k,m)}$ with 0 for those k branches that are 'full', i.e have reached the limit for number of children per branch. We assign $S[1].label$ a location in the *Tree-CNN* depending on its *value*. After that, we remove the column corresponding to that class from $L(k, m)$, we check for "full" branch nodes, and modify $L(k, m)$ for those output nodes. Finally we generate the ordered list S, and again apply our conditions on the new $S[1]$ to determine where it is added to the root node. This is done iteratively till all new classes are assigned a location under the root node.

The pseudo-code is outlined in Algorithm 2. We also illustrate a toy example of incremental learning in Tree-CNN with Fig. 3.2. The network starts as a single CNN that can classify 3 classes, C_1, C_2, C_3. We want to increase the network capability by adding 3 new classes. In the first incremental learning stage, the softmax likelihood table L is generated, as shown in the figure. C_4 and C_5 are added to the leaf nodes containing C_1 and C_2 respectively, converting them into branch nodes B_1 and B_2, as per condition (i). For C_6, the 3 likelihood values are $v_1 = 0.34, v_2 = 0.33, v_3 = 0.33$. It satisfies neither condition (i) nor condition (ii), thus it is added as a new node to the root, as per condition (iii). Again, as new information is available, we want the Tree-CNN to be able to recognize 2 new image classes, C_7, and C_8. Both the new classes satisfy $v_1 - v_2 > \alpha(= 0.1)$. Thus, both the classes are added to B_1. While this example is for a two level Tree-CNN, the algorithm can potentially be extended to deeper Tree-CNN models.

To create deeper Tree-CNN models, once the "Grow-Tree" algorithm is completed for the M classes at the root node, one can move to the next level of the tree. The same process is applicable on the child nodes that now have new classes to be added to them. The decision on how to grow the tree is semi-supervised: the algorithm itself decides how to grow the tree, given the constraints by the user. We can limit parameters such as maximum children for a node, maximum depth for the tree, etc. as per our system requirements.

Once the new classes are allotted locations in the tree, supervised gradient descent based training is performed on the modified/new nodes. This saves us from modifying the whole network, and only affected portions of the network require retraining/fine-tuning. At every incremental learning stage, the root node is trained on all the available data as it needs to learn to classify all the objects into the new branches. During inference, a branch node is

Algorithm 2 Grow Tree-CNN

L = Likelihood Matrix
$maxChildren$ = max. number of children per branch node
$RootNode$ = Root Node of the Tree-CNN
procedure GrowTree(L, Node)
    $S = GenenerateS(L, Node, maxChildren)$
    while S is not Empty **do**
        Get attributes of the first object
        $[label, value, node] = GetAttributes(S[1])$
        if $value[1] - value[2] > \alpha$ **then**
            The new class has a strong preference for n_1
            Adds $label$ to $node[1]$
            $RootNode = AddClasstoNode(RootNode, label, node[1])$
        else
            if $value[2] - value[3] > \beta$ **then**
                The new class has similar strong preference n_1 and n_2
                $Merge = CheckforMerge(Node, node[1], node[2])$
                $Merge$ is $True$ only if $node[2]$ is a leaf node, and,
                the # of children of $node[1]$ less than $maxChildren - 1$
                if $Merge$ **then**
                    Merge $node[2]$ into $node[1]$
                    $RootNode = MergeNode(RootNode, node[1], node[2])$
                    $RootNode = AddClasstoNode(RootNode, label, node[1])$
                else
                    Add new class to the smaller output node
                    $sNode$ = Node with lesser children $(node[1], node[2])$
                    $RootNode = AddClasstoNode(RootNode, label, sNode)$
                end if
            else
                Add new class as a new Leaf node to Root Node
                $RootNode = AddNewNode(RootNode, label)$
            end if
        end if
        Remove the columns of the added class from L
        Remove the rows of "full" nodes from L
        Regenerate S
        $S = GenenerateS(L, Node, maxChildren)$
    end while
end procedure

activated only when the root node classifies the input to that branch node. If an incorrect classification happens at Root Node, for example it classifies an image of a car into the "Animal Node" (CIFAR-10 example, Sec 4.1), irrespective of what the branch node classifies it as, it would still be an incorrect classification. Hence we only train the branch node with the classes it has been assigned to. If there is no change in the branch node's look up table at an incremental learning stage, it is left as is.

Handling input labels inside the Tree-CNN

The dataset available to the user will have unique labels assigned to each of it's object classes. However, the root and branch nodes of the Tree-CNN tend to group/merge/split these classes as required by the algorithm. To ensure label consistency, each node of the Tree-CNN maintains it's own "LabelsTransform" lookup table. For example, when a new class is added to one of the pre-existing output nodes of a root node, the lookup table is updated with new class being assigned to that output node. Similarly when a new class is added as a new node, the class label and the new output node is added as a new entry to the lookup table. Every class is finally associated with a unique leaf node, hence leaf nodes do not require a look up table. Whenever two nodes are merged, the node with lower average softmax value (say, node A) gets integrated with the node with the higher average softmax value (say, node B) for the new class in consideration. If the two softmax values are equal, it is chosen at random. At the root node level, the lookup table is modified as follows: The class labels that were assigned to node A, will now be assigned to node B. The look up table of merged node B will add these class labels from node A as new entries and assign them to new leaf nodes.

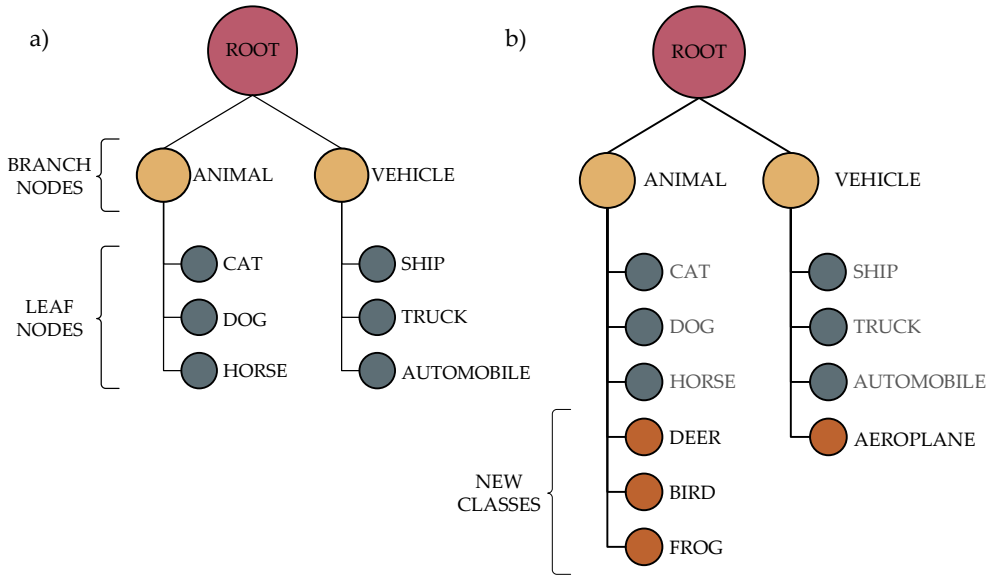


Figure 3.3. : Graphical representation of Tree-CNN for CIFAR-10 **a)** before incremental learning, **b)** after incremental learning

3.4 The Experimental Setup

3.4.1 Adding Multiple New Classes (CIFAR-10)

Dataset

CIFAR-10 dataset [60], having 10 mutually exclusive classes, was used for this experiment. The network is first trained on 6 classes, and then learns the remaining 4 classes in the next learning stage.

The Network Initialization

The *Tree-CNN* for CIFAR-10 starts out as a two level network with a root node with two branch nodes as shown in Fig 3.3. The six initial classes of CIFAR-10 are grouped into "Vehicles" and "Animals" and the CNN (Table 3.1) at the root is trained to classify input images into these two categories. Each of the two branch nodes has a CNN (Table 3.2) that does finer classification into leaf nodes. Fig. 3.3a) represents the initial model of *Tree-CNN A*. This experiment illustrates, that given provided a 2-level *Tree-CNN*, how the learning model can add new classes. The root node achieves a testing accuracy of 98.73%,

Table 3.1. : Root Node Tree-CNN (CIFAR-10)

Input 32×32×3
CONV-1 64 5×5 ReLU
[2 2] Max Pooling
CONV-2 128 3×3 ReLU Dropout 0.5 128 3×3 ReLU
[2 2] Max Pooling
FC 8192×512 ReLU Dropout 0.5 512×128 ReLU Dropout 0.5 128×2 ReLU
Softmax Layer

Table 3.2. : Branch Node Tree-CNN (CIFAR-10)

Input 32×32×3
CONV-1 32 5×5 ReLU
[2 2] Max Pooling
Dropout 0.25
CONV-2 64 5×5 ReLU
[2 2] Max Pooling
Dropout 0.25
CONV-3 64 3×3 ReLU
[2 2] Avg Pooling
Dropout 0.25
FC 1024×128 ReLU Dropout 0.5 128×N ReLU (N = # of Classes)
Softmax Layer

Table 3.3. : Network B

Input 32×32×3
CONV-1 64 3×3 ReLU Dropout 0.5 64 3×3 ReLU
[2 2] Max Pooling
CONV-2 128 3×3 ReLU Dropout 0.5 128 3×3 ReLU
[2 2] Max Pooling
CONV-3 256 3×3 ReLU Dropout 0.5 256 3×3 ReLU
[2 2] Max pooling
CONV-4 512 3×3 ReLU Dropout 0.5 512 3×3 ReLU
[2 2] Avg Pooling
FC 2048×1024 ReLU Dropout 0.5 1024×1024 ReLU Dropout 0.5 1024×N (N= # of Classes)

while the branch nodes, "Animals" and "Vehicles", achieve 86% and 94.43% testing accuracy respectively. Overall, the network achieves a testing accuracy of 89.10%.

Incremental Learning

The remaining four classes are now introduced as the new learning task. 50 images per class (10% of the training set) are selected at random , and shown to the root node. We obtain the L matrix, which is a 2×4 matrix with each element $l_{ij} \in (0, 1)$. The 1^{st} row of the matrix indicates the softmax likelihood of each of the 4 classes as being classified

as “Vehicles”, while the second row presents the same information for “Animals”. In this experiment, α is set at 0 (Algorithm 2), and the network is bound to take only one action: add the new class to one of the two child nodes. The branch node with higher likelihood value adds the new class to itself. The *Tree-CNN* before and after addition of these 4 classes is shown in Fig. 3.3 .

Once the new classes have been assigned locations in the Tree-CNN, we begin the re-training of the network. The root node is re-trained using all 10 classes, divided into to subclasses. The branch node ”animal” is retrained using training data from 6 classes, 3 old and 3 new added to it. Similarly, branch node ”vehicles” is retrained with training data from 4 classes, 3 old, 1 new.

3.4.2 Sequentially Adding Multiple Classes (CIFAR-100)

Table 3.4. : Root Node Tree-CNN (CIFAR-100)

Input 32×32×3
CONV-1 64 5×5 ReLU
[2 2] Max Pooling
CONV-2 128 3×3 ReLU Dropout 0.5 128 3×3 ReLU
[2 2] Max Pooling
CONV-3 256 3×3 ReLU Dropout 0.5 256 3×3 ReLU
[2 2] Avg Pooling
FC 4096×1024 ReLU Dropout 0.5 1024×1024 ReLU Dropout 0.5 1024×N (N = # of Children)

Table 3.5. : Branch Node Tree-CNN (CIFAR-100)

Input 32×32×3
CONV-1 32 5×5 ReLU
[2 2] Max Pooling
Dropout 0.25
CONV-2 64 5×5 ReLU
[2 2] Max Pooling
Dropout 0.25
CONV-3 64 3×3 ReLU Dropout 0.5 64 3×3 ReLU
[2 2] Avg Pooling
FC 1024×512 ReLU Dropout 0.5 512×128 ReLU Dropout 0.5 128×N (N = # of Children)

Dataset

The dataset, CIFAR-100 [60], has 100 classes, 500 training and 100 testing images per class. The 100 classes are randomly divided into 10 groups of 10 classes each and organized in a fixed order (A.1). These groups of classes are introduced to the network incrementally.

The Network Initialization

We initialize the *Tree-CNN* as a root node with 10 leaf nodes. The root node, thus comprises of a CNN (Table 3.4), with 10 output nodes. Initially this CNN is trained to classify the 10 classes belonging to group 0 of the incremental CIFAR-100 dataset (A.1). In subsequent learning stages, as new classes get grouped together under same output nodes, the network adds branch nodes. The DCNN model used in these branch nodes is given in Table 3.5. The branch node has a higher chance of over-fitting than the root node as the dataset per node shrinks in size as we move deeper into the tree. Hence we introduce more dropout layers to the CNNs at these nodes to enhance regularization.

Incremental Learning

The remaining 9 groups, each containing 10 classes is incrementally introduced to the network in 9 learning stages. At each stage, 50 images belonging to each class are shown to the root node and a likelihood matrix L is generated. The columns of the matrix are used to form an ordered set S, as described in Sec. 3.3.2 . For this experiment, we applied the following constraints to the Algorithm 2:

- Maximum depth of the tree is 2.
- We set $\alpha = 0.1$ and $\beta = 0.1$.
- Maximum number of child nodes for a branch node is set at 5, 10, 20 for the three test cases: *Tree-CNN-5*, *Tree-CNN-10*, and *Tree-CNN-20* respectively.

At every learning stage, once the new classes have been assigned the location in the Tree-CNN, we update the corresponding branch and root CNNs by retraining them on the

combined dataset of old and new classes added to them. The branch nodes to which new children have not been added are left untouched.

3.4.3 Benchmarking

There is an absence of standardized benchmark protocol for incremental learning, which led us to use a benchmarking protocol similar to one used in iCaRL [51]. The classes of the dataset are grouped and arranged in a fixed random order. At each learning stage, a selected set of classes would be introduced to the network. Once training is completed for a particular learning stage, the network would be evaluated on all the classes it has learned so far and the accuracy is reported.

Baseline Network

To compare against the proposed *Tree-CNN*, we defined a baseline network (Network B) with a complexity level similar to two stage *Tree-CNN*. The network is has a VGG-net [61] like structure with 11 layers. It has 4 convolutional blocks, each block having 2 sets of 3×3 convolutional kernels (Table 3.3).

Fine-tuning the baseline network using old + new data

The baseline network is trained in incremental stages using fine-tuning. The new classes are added as new output nodes of the final layer and 5 different fine tuning strategies have been used. Each method retrains/fine-tunes certain layers of the network. While fine tuning, all of the available dataset is used, both old data and new data. It is assumed that the system has access to all the data that has been introduced so far. As listed below, we set 5 different depths of back-propagation when retraining with the incremental data and the old data.

- B:I [FC]
- B:II [FC + CONV-1]
- B:III [FC + CONV-1 + CONV-2]
- B:IV [FC + CONV-1 + CONV-2 + CONV-3]

- B:V [FC + CONV-1 + CONV-2 + CONV-3 + CONV-4] (equivalent to training a new network with all the classes)

Evaluation Metrics

We compare *Tree-CNN* against retraining Network B on two metrics: Testing Accuracy, and Training Effort , which is defined as

Training Effort = $\sum_{nets}$ (total number of weights $\times$ total number of training samples)

Training Effort attempts to capture the number of weight updates that happen per training epoch. As batch size and number of training epochs is kept the same, the product of the number of weights and the number of training samples used can provide us with the measure of the computation cost of a learning stage. For *Tree-CNN* the training effort of each of the nodes (or nets) is summed together. Whereas, for network B, it is just one node/neural network, and for each of the cases (*B:I-B:V*), we simply sum the number of weights in the layers that are being retrained and multiply it with total number of training samples available at a learning stage to calculate the Training Effort.

3.4.4 The Training Framework

We used MatConvNet [62], an open-source Deep Learning toolbox for MATLAB [63], for training the networks. During training, data augmentation was done by flipping the training images horizontally at random with a probability of 0.5 [64]. All images were whitened and contrast normalized [64]. The activation used in all the networks is rectified linear activation ReLU, $\sigma(x) = max(x, 0)$. The networks are trained using mini-batch stochastic gradient descent with fixed momentum of 0.9. Dropout [38] is used between the final fully connected layers, and between pooling layers to regularize the network. We also employed batch-normalization [65] at the output of every convolutional layer. Additionally, a weight decay $\lambda = 0.001$ was set to regularize each model. The weight decay helps against overfitting of our model. The final layer performs softmax operation on the output of the nodes to generate class probabilities. All CNNs are trained for 300 epochs. The learning rate is kept at 0.1 for first 200 epochs, then reduced by a factor of 10 every 50 epochs.

3.5 Results

3.5.1 Adding multiple new classes (CIFAR-10)

Table 3.6. : Training Effort and Test Accuracy comparison of Tree-CNN against Network B for CIFAR-10

	B:I	B:II	B:III	B:IV	B:V	Tree-CNN
Testing Accuracy	78.37	85.02	88.15	90.00	90.51	86.24
Normalized Training Effort	0.40	0.85	0.96	0.99	1	0.60

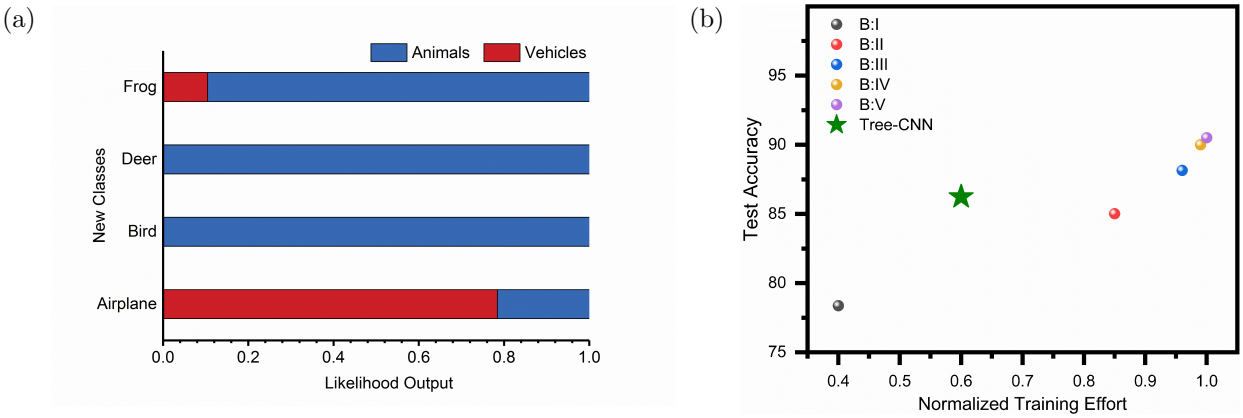


Figure 3.4. : Incrementally learning CIFAR-10: 4 New classes are added to Network B and *Tree-CNN*. Networks *B:I* to *B:V* represent 5 increasing depths of retraining for Network B. (a) The softmax likelihood output at the root node for the two branches. (b) Testing Accuracy vs Normalized Training Effort for *Tree-CNN* and networks *B:I* to *B:V*

We initialized a *Tree-CNN* that can classify six classes (Fig. 3.3a). It had a root node and two branch nodes. The sample images from the 4 new classes generated the softmax likelihood output at root node as shown in Fig. 3.4a. Accordingly, the new classes are added to the two nodes, and the new *Tree-CNN* is shown in Fig. 3.3b. In Table 3.6, we report the test accuracy and the training effort for the 5 cases of fine-tuning network *B* against our *Tree-CNN* for CIFAR-10. We observe that retraining only the FC layers of baseline network (*B:I*) requires the least training effort, however, it gives us the lowest accuracy of 78.37%. And as more classes are introduced, this method causes significant loss in accuracy, as shown with CIFAR-100 (Fig. 3.6b). The *Tree-CNN* has the second lowest normalized

training effort, $\sim 40\%$ less than *B:V*, and $\sim 30\%$ less than *B:II*. At the same time, *Tree-CNN* had comparable accuracy to *B:II* and *B:III*, while just being less than the ideal case *B:V* by a margin of 3.76%. This accuracy vs training effort trade-off is presented in Fig. 3.4b, where it is clearly visible that *Tree-CNN* provided the most optimal solution for adding the 4 new classes.

3.5.2 Sequentially adding new classes (CIFAR-100)

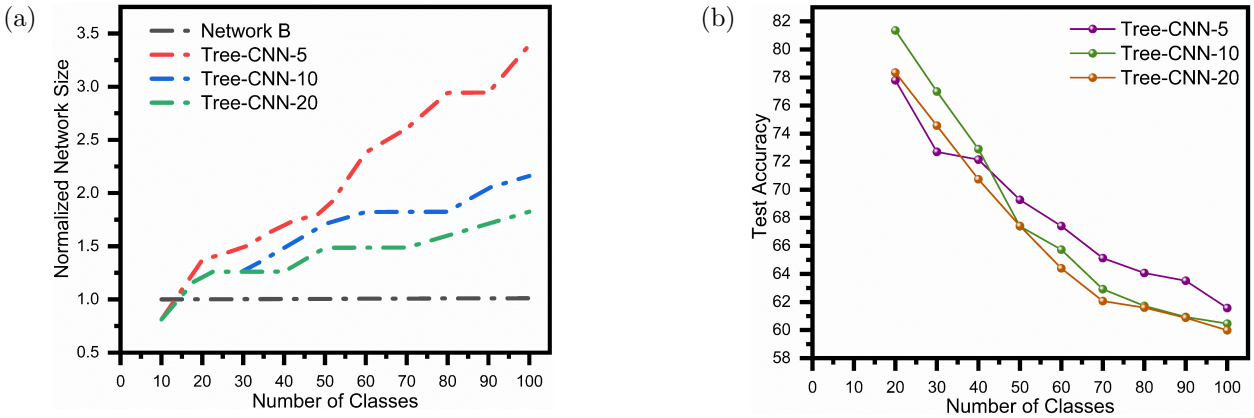


Figure 3.5. : *Tree-CNN*: Effect of varying the maximum number of children per branch node (*maxChildren*) as new classes are added to the models (CIFAR-100) (a) Network size (b) Test Accuracy

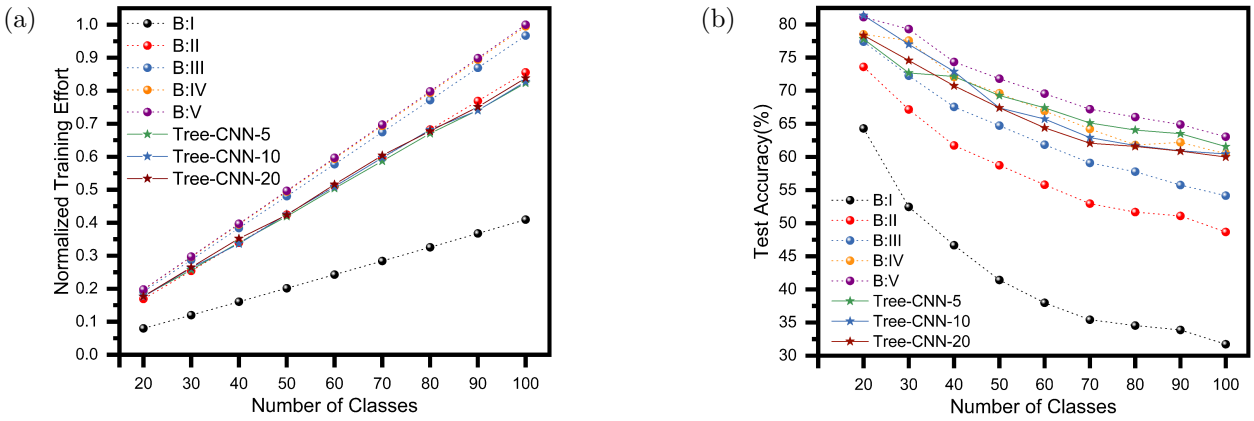


Figure 3.6. : CIFAR-100: New classes are added to Network B and *Tree-CNNs* in batches of 10. Networks B:I to B:V represent 5 increasing depths of retraining for Network B (a) Training effort for every learning stage (Table A.1) (b) Testing Accuracy at the end of each learning stage (Table A.2)

We initialized a root node that can classify 10 classes, i.e. has 10 leaf nodes. Then, we incrementally grew the *Tree-CNN* for 3 different values of maximum children per branch node (*maxChildren*), namely 5, 10, and 20. We label these 3 models as *Tree-CNN-5*, *Tree-CNN-10* and *Tree-CNN-20* respectively. At the end of 9 incremental learning stages, the root node of *Tree-CNN-5* had 23 branch nodes and 3 leaf nodes. Whereas, the root node of *Tree-CNN-10* has 12 branch nodes and 5 leaf nodes. As expected, the root node of *Tree-CNN-20* had least number of child nodes, 9 branch nodes and 3 leaf nodes. The final hierarchical structure of the *Tree-CNNs* can be found in A.2, Fig. A.1-A.3.

We observe that as new classes are added, the *Tree-CNNs* grow in size by adding more branches (Fig. 3.5a). The size of Network *B* remains relatively unchanged, as only additional output nodes are added, which translates to a small fraction of new weights in the final layer. *Tree-CNN-5* almost grows 3.4× the size of Network *B*, while *Tree-CNN-10* and *Tree-CNN-20* reach 2.2× and 1.8× the baseline size, respectively. The training effort for the 3 *Tree-CNNs* was almost identical, within 1e-2 margin of each other (Fig. 3.6a), over the 9 incremental learning stages. As *maxChildren* is reduced, the test accuracy improves, as observed in Fig. 3.5b. If *maxChildren* is set to 1, we obtain a situation similar to test case *B:V*, where every new class is just a new output node.

We compare the training effort needed for the *Tree-CNNs* against the 5 different fine-tuning cases of Network *B* over the 9 incremental learning stages in Fig. 3.6a. We normalized the training effort by dividing all the values with the highest training effort. i.e. *B:V*. For all the models, the training effort required at a particular learning stage was greater than the effort required by the previous stage. This is because we had to show images belonging to old classes to avoid "catastrophic forgetting". The *Tree-CNNs* exhibit a lower training effort than 4 fine-tuning test cases, *B:II* - *B:V*. the test case *B:I* has a significantly lower training effort than all the other cases, as it only retrains the final fully connected layer. However, it suffers the worst accuracy degradation over the 9 learning stages (Fig. 3.6b). This shows that only retraining the final linear classifier, i.e. the fully connected layer, is not sufficient. We need to train the feature extractors, i.e. convolutional blocks, as well on the new data.

While *B:I* is the worst performer in terms of accuracy, Fig. 3.6b shows that all the networks suffer from some accuracy degradation with increasing number of classes. *B:V*

provides the baseline accuracy at each stage, as it represents a network fully trained on all the available data at that stage. The three *Tree-CNNs* perform almost at par with *B:IV*, and outperform all other variants of network *B*. From Fig. 3.6, we can conclude that *Tree-CNNs* offer the most optimal trade-off between training effort and testing accuracy. This is further illustrated in Fig. 3.7a, where we plot the average test accuracy and average training effort over all the learning stages.

Table 3.7. : Test Accuracy over all 100 classes of CIFAR-100

Model	Final Test Accuracy (%)	Average Test Accuracy(%)
B:V	63.05	72.23
Tree-CNN-5	61.57	69.85
Tree-CNN-10	60.46	69.53
Tree-CNN-20	59.99	68.49
iCarl (Rebuffi, et al. 2017) [51]	49.11	64.10
LwF (Li, et al. 2017) [48], [51]	25.00	44.49
HD-CNN (Yan, et al. 2015) [57]	67.38	N/A
Hertel, et al. 2015 [66]	67.68	N/A

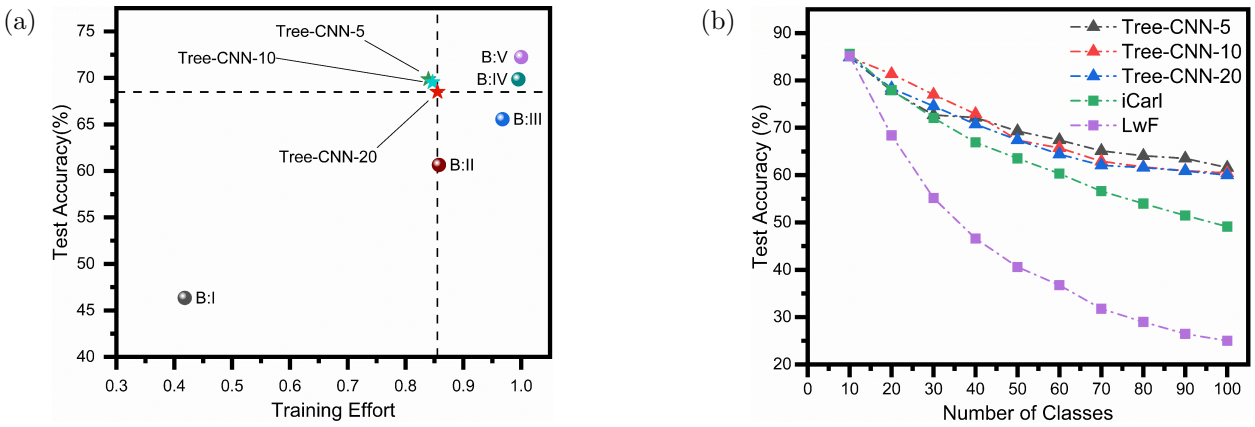


Figure 3.7. : The performance of *Tree-CNN* compared with a) Fine-tuning Network *B* b) Other incremental learning methods [48], [51]

We compare our model against two works on incremental learning, 'iCaRL'[51] and 'Learning without Forgetting' [48] as shown in Fig. 3.7b. We use the accuracy reported in [51] for CIFAR-100, and compare it against our method. For 'LwF', a ResNet-32 [67] is retrained exclusively on new data at every stage. Hence it suffers the most accuracy degra-

dation. In 'iCarl' [51], a ResNet-32 is retrained with new data and only 2000 samples of old data (called 'exemplars') at every stage. It is able to recover a good amount performance, compared to 'LwF' but still falls short of state-of-the-art by ≈18%. *Tree-CNNs* yield 10% higher accuracy than 'iCaRL' and over 50% higher accuracy than 'Learning without Forgetting' (LwF). This shows that our learning method using the hierarchical structure is more resistant to catastrophic forgetting as new classes are added. *Tree-CNNs* are able to achieve near state-of-the-art accuracy for CIFAR-100 as illustrated in Table 3.7. While the second column reports the final accuracy, the third column reports the average accuracy of the incremental learning methods where new classes are added in batches of 10.

An interesting thing to note was similar looking classes, that were also semantically similar, were grouped under the same branches. At the end of the nine incremental learning stages, certain similar objects grouped together is shown in Fig. 3.8 for *Tree-CNN-10*. While there were some groups that had object sharing semantic similarity as well, there were odd groups as well, such as Node 13 as shown in Fig. 3.8. This opens up the possibility of using such a hierarchical structure for finding hidden similarity in the incoming data.

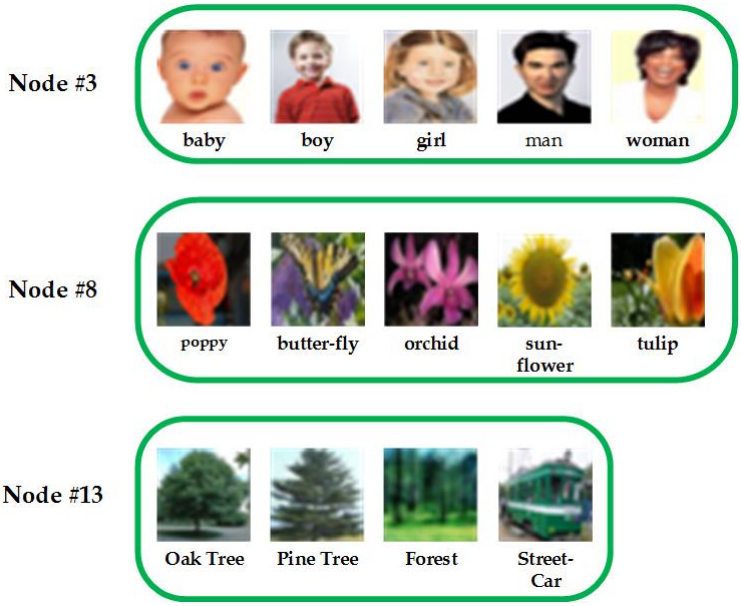


Figure 3.8. : Examples of groups of classes formed when new classes were added incrementally to *Tree-CNN-10* in batches of 10 for CIFAR-100

3.6 Discussion

The motivation of this work stems from the idea that subsequent addition of new image classes to a network should be easier than retraining the whole network again with all the classes. We observed that each incremental learning stage required more effort than the previous, because images belonging to old classes needed to be shown to the CNNs. This is due to the inherent problem of "catastrophic forgetting" in deep neural networks. Our proposed method offers the best trade-off between accuracy and training effort when compared against fine-tuning layers of a deep network. It also achieves better accuracy, much closer to state-of-the-art on CIFAR-100, as compared to other works, 'iCarl' [51] and 'LwF' [48]. The hierarchical node-based learning model of the *Tree-CNN* attempts to confine the change in the model to a few nodes only. And, in this way, it limits the computation costs of retraining, while using all the previous data. Thus, it can learn with lower training effort than fine-tuning a deep network, while preserving much of the accuracy. However, the *Tree-CNN* continues to grow in size over time, and the implications of that on memory requirements needs to be investigated. During inference, a single node is evaluated at a time, thus the memory requirement per node inference is much lower than the size of the entire model. *Tree-CNN* grows in a manner such that images that share common features are grouped together. The correlation of the semantic similarity of the class labels and the feature-similarity of the class images under a branch is another interesting area to explore. The *Tree-CNN* generates hierarchical grouping of initially unrelated classes, thereby generating a label relation graph out of these classes[33]. The final leaf nodes, and the distance between them can also be used as a measure of how similar any two images are. Such a method of training and classification can be used to hierarchically classify large datasets. Our proposed method, Tree-CNN, thus offers a better learning model that is based on hierarchical classifiers and transfer learning and can organically adapt to new information over time.

4. DEEP STOCHASTIC NEURAL NETWORKS

With the increasing complexity of the tasks accomplished by Deep Learning, the computational cost of these models have also risen enormously. In the modern era of ubiquitous Artificial intelligence (AI) and smart devices, there is a need to enable AI tasks in highly constrained low power and low memory environments. The growing demand of Deep Learning in such everyday applications has thus accelerated the search for fast and power efficient implementations of these networks.

Achieving speed up and power efficiency during inference can be approached by efficient hardware-software co-design principles. To that effect, researchers are investigating several methods to build smaller neural network models with simpler computations [68]. Novel network architectures, such as SqueezeNet [69] and MobileNet [70] have been proposed that employ alternate convolution techniques such as depthwise convolution using 1×1 filters for repeated squeezing and expansion of feature maps. On the other hand, model pruning techniques selectively remove redundant convolutional filters to achieve optimal filter bank width [71].

While these methods focus on changing the model architecture such as the number of layers and filters, another way of model compression is quantization of the weights and inputs. Traditionally weights of a neural network are stored as 32/64 bit floating point values. Reducing bit precision of these weights can help minimize the model size. The inherent redundancy and resiliency of neural networks towards approximate computations can potentially help us quantize these networks without significant loss in performance [72]. Using binary neuron activations, such as 0/1 or -1/1, one can simplify weight-input multiplications to just additions [73]. When both weights and activations are binary, multiply and accumulate operations can be replaced by simple bitwise operations as shown in BinaryNets [74] and XNOR-Nets [75]. Binary activations inside a deep learning model create a binary communication channel between layers, which in a way mimics the spike based communication observed in biological neurons. Such networks fall under the umbrella of the third generation of neural networks, called spiking neural networks that are uniquely characterized by the spike-like nature of their inter layer communication.

The communication between biological neurons is often noisy with high frequency aberrations. Neuroscientists have often pondered on the impact of such probabilistic interaction in the brain, whether it enables greater functionality. Several works [35], [36] indicate that cortical microcircuits of pyramidal neurons perform Bayesian computations, and their switching behavior can be modelled by a sigmoid [11]. In this work we design deep SNNs with such stochastic sigmoid neurons. We also explore binary neurons that follow a linear (hardtanh) switching probability dependence with the input.

In the context of emulating the functionality of biological systems, designing custom hardware to perform the operations specific to machine learning workloads more efficiently than general-purpose systems has been heavily explored. This has led researchers to venture beyond traditional Von-neumann computing. Such systems are not only massively parallel but also capable of compact and efficient implementations of the fundamental units of the neural networks, namely neurons and synapses. This new paradigm of computing is inspired from the mammalian brain.

Various technologies based on both CMOS and non-volatile memory (NVM) devices have been used to explore the aforementioned stochastic characteristics of binary neurons. In CMOS technology, such stochasticity is introduced through biased random number generator (RNG) circuits. Standard RNG circuits, however, require a number of transistors which make CMOS often expensive in terms of area, despite being able to be conveniently implemented. NVM devices, on the other hand, can emulate neuronal functionalities using single, isolated devices which enable compact implementation. Moreover, the inherent stochasticity in these devices make them quite amenable towards implementing stochastic binary neurons. To that effect, recent work on Phase-Change Memory (PCM) technology has experimentally demonstrated stochastic firing behavior in neuronal devices [76]. The variability in Resistive Random Access Memory (RRAM) can also be leveraged to enable stochasticity. However, it is to be noted, that for both RRAM and PCMs, the stochasticity is often not controllable. In contrast, spintronic devices such as magnetic tunnel junctions (MTJ) offer current controlled stochasticity which has been used to emulate stochastic behavior in neurons [11]. Moreover, device proposals leveraging the high spin-injection efficiency of the spin-hall effect (SHE) by stacking the MTJs on top of a heavy metal layer can lead to lower write currents and higher

energy-efficiency. MTJ-based proposals of stochastic neurons thus promise to be compact and energy-efficient.

Although MTJ provides a useful solution toward energy-efficient implementations of stochastic spiking neural networks, studies have been limited to simple pattern recognition tasks on small datasets using shallow networks [12], [13]. However, for real world deployment of such energy-efficient systems, it is necessary to investigate the scalability of such spiking neural networks with stochastic binary activations. In this work [77] we design an 8 layer and a 16 layer VGG network [61] with different kinds of binary activations, both deterministic and stochastic and observe their performance on large datasets, CIFAR10 and CIFAR100 [60].

In Sec. 4.1, we define the different kinds of binary activations and the two different weight binarizing techniques. The simulation framework is described in Sec. 4.2. Next, we discuss our key findings while training these stochastic spiking neural networks in Sec. 4.3. We analyze the hardware implications of such neural networks in 4.4. Finally we conclude in Sec. 4.5 the pros and cons of implementing such networks on existing as well as emerging hardware platforms.

4.1 Training Deep Spiking Neural Networks with Binary Stochastic Activation

A spiking neural network is characterized by its event driven binary communication. Each neuron produces a spike only when certain conditions are met, otherwise remains idle. The presence/absence of spikes can be encoded as a binary value. While advanced spiking neural network models incorporate more bio-plausible features like Integrate and Fire (IF) neurons, spike timing dependent plasticity (STDP) etc., an artificial neural network (ANN) with binary activation has the most fundamental property of SNNs, i.e. binary communication [78]. One can potentially train these ANNs with backpropagation and stochastic gradient descent [79]. However, in doing so, one has to use a smooth approximation for the discontinuous gradient of the binary neuron. In the following subsection, we discuss the various types of binary neurons and their gradients during training.

4.1.1 Binary Stochastic Activations

A binary neuron has an output that can be represented by two distinct states. While there can be many ways to encode the binary states, in this work we use the two most common ones, unipolar (0/1) and bipolar (+1/ − 1) representations. The input to the neuron is a weighted sum of all the neurons connected to it from the previous layer, and this input is non-binary in a neural network. The neuron can use a deterministic or a stochastic method to convert its input to a binary output.

For unipolar neurons, we investigate the binary sigmoid neuron, as proposed in [13]. In [13], the ANN was initially trained with a sigmoid neuron, and at inference, the sigmoid was replaced with a stochastic sigmoid spiking neuron. Such a neuron outputs a 1, i.e. a spike with a probability determined by the sigmoid of its input (Eq. 4.1). The network would then be evaluated over multiple iterations.

$$a = \begin{cases} 1 \text{ with } p_1 = sigmoid(z) = \frac{1}{1+e^{-z}} \\ 0 \text{ with } p_0 = 1 - p_1 = 1 - sigmoid(z) \end{cases} \tag{4.1}$$

We eliminate the conversion step by directly training the network with stochastic sigmoid (*stochSigmoid*) neuron. By doing so, we remove the need to evaluate the network over multiple iterations. The network only needs to be evaluated once to obtain the class label. We also evaluate our network for a deterministic version of the binary sigmoid neuron, and we refer to it as *detSigmoid* going forward.

$$a = \begin{cases} 1, sigmoid(z) \geq 0.5 \\ 0, sigmoid(z) < 0.5 \end{cases} \tag{4.2}$$

In both cases, the derivative of the neuron, $\frac{\partial a}{\partial z}$ is computed as the derivative of a sigmoid.

$$\frac{\partial a}{\partial z} = \frac{\partial sigmoid(z)}{\partial z} = \frac{e^{-z}}{(1+e^{-z})^2} \tag{4.3}$$

Next, we use the hardtanh function to model the bipolar binary neuron, as seen in BinaryNet [74]. Here z is the input to the neuron, and a is the output. First we compute $y = hardtanh(z)$ which is given by,

$$y = \begin{cases} min(z, 1), & z \geq 0 \\ max(z, -1), & z < 0 \end{cases} \tag{4.4}$$

Then, y can be binarized in a deterministic or stochastic manner. In case of deterministic binarization,

$$a = \begin{cases} 1, & y \geq 0 \\ -1, & y < 0 \end{cases} \tag{4.5}$$

For stochastic binarization, a random number r is generated from the uniform distribution $U(0, 1)$, and y is compared against it to generate z. This implies that the probability (p) that z is $1(-1)$ is $p_1 = \frac{(y+1)}{2}$ $(p_{-1} = 1 - \frac{(y+1)}{2})$.

$$a = \begin{cases} 1, & \frac{(y+1)}{2} \geq r \\ -1, & \frac{(y+1)}{2} < r \end{cases} \tag{4.6}$$

During backpropagation, the derivative of the neuron $\frac{\partial a}{\partial z}$ is computed as the derivative of hardtanh. Going forward, we refer to deterministic binary hardtanh neuron as *detHardtanh*, and its stochastic counterpart as *stochHardtanh*

$$\frac{\partial a}{\partial z} = \begin{cases} 0, & z > 1 \\ 1, & -1 \leq z \leq 1 \\ 0, & z < -1 \end{cases} \tag{4.7}$$

In Table 4.1, we have listed all the neuron models used in this work, and their derivatives.

Table 4.1. : Neuron Models

Name	Type	Output	$a = f(z)$	$\partial a/\partial z$
ReLU	N/A	$[0, \infty)$	$max(0, z)$	1 if $z > 0$ else 0
detHardtanh	deterministic	$1, -1$	$\begin{cases} 1, & z \geq 0 \\ -1, & z < 0 \end{cases}$	$\begin{cases} 0, & z > 1 \\ 1, & -1 \leq z \leq 1 \\ 0, & z < -1 \end{cases}$
stochHardtanh	stochastic	$1, -1$	$\begin{cases} 1 \text{ with } p_1 = (y+1)/2 \\ -1 \text{ with } p_0 = 1 - p_1 \end{cases}$	$\begin{cases} 0, & z > 1 \\ 1, & -1 \leq z \leq 1 \\ 0, & z < -1 \end{cases}$
detSigmoid	deterministic	$1, 0$	$\begin{cases} 1, sigmoid(z) \geq 0.5 \\ 0, sigmoid(z) < 0.5 \end{cases}$	$\frac{\partial sigmoid(z)}{\partial z} = \frac{e^{-z}}{(1+e^{-z})^2}$
stochSigmoid	stochastic	$1, 0$	$\begin{cases} 1 \text{ with } p_1 = sigmoid(z) = \frac{1}{1+e^{-z}} \\ 0 \text{ with } p_0 = 1 - p_1 = 1 - sigmoid(z) \end{cases}$	$\frac{\partial sigmoid(z)}{\partial z} = \frac{e^{-z}}{(1+e^{-z})^2}$

4.1.2 Weight Quantization

In the Sec 4.1.1, we discussed the various kinds of binary activations. While binary activations simplify Multiply-and-Accumulate (MAC) operations to simple additions, we can further reduce computations by binarizing the weights, thereby reducing compute-intensive floating point multiplications to bitwise operations. We use the following two weight quantization techniques:

- BinaryNet [74] : Real valued weights are constrained to $[1, -1]$ by clipping weights outside of the range. Then they were binarized by $w^b = Sign(w)$. Except the first layer, all weights were binarized.
- XNOR-Net [75] : These networks perform scaled binarization, where a convolution filter is binarized as $W^b = ||W||_1 Sign(W)$. The first and the last layer of weights are kept as full precision. Also the first and the last activation layers are kept as ReLU (i.e. full precision).

4.2 Experiments

4.2.1 Simulation Framework

For our simulations, we used Torch7 [80], a Lua based scientific computing framework that supports a wide variety of machine learning algorithms with GPU compatible implementations. The simulations were run on a single GeForce Titan XP GPU. CIFAR-10 and CIFAR-100 [60] were used to evaluate the performance of the deep stochastic SNNs. Both the datasets are composed of 3 channel RGB images of size 32×32 with $50,000$ training samples and $10,000$ test samples. CIFAR-10 has 10 classes, each class having 5000 training and 1000 test images. CIFAR-100 has 100 classes, with each class having 500 training and 100 test images.

Table 4.2. : Network Architecture

(a) VGG-9	(b) VGG-16
3×3 Conv 3, 128	3×3 Conv 3, 64
3×3 Conv 128, 128	3×3 Conv 64, 64
Max Pool (2,2)	Max Pool (2,2)
3×3 Conv 128, 256	3×3 Conv 64, 128
3×3 Conv 256, 256	3×3 Conv 128, 128
Max Pool (2,2)	Max Pool (2,2)
3×3 Conv 256, 512	3×3 Conv 128, 256
3×3 Conv 512, 512	3×3 Conv 256, 256
Max Pool (2,2)	3×3 Conv 256, 256
4096×1024 Linear	Max Pool (2,2)
1024×1024 Linear	3×3 Conv 256, 512
1024×10 Linear	3×3 Conv 512, 512
	3×3 Conv 512, 512
	Max Pool (2,2)
	3×3 Conv 512, 512
	3×3 Conv 512, 512
	3×3 Conv 512, 512
	Max Pool (2,2)
	512×4096 Linear
	4096×4096 Linear
	4096×4096 Linear

4.2.2 Experiments with CIFAR-10

For CIFAR-10 dataset, we selected a 9 layer VGG network [61], and the complete network architecture is provided in Table 4.2(a). We trained the network for 3 different weight quantizations and 5 different neuron activation functions, as described in Sec. 4.1. At every layer, except the last layer, batch-normalization [65] is applied. For training deeper networks, particularly with saturating functions like sigmoid and hardtanh, batch-normalization is needed to ensure the neurons operate in the linear region, and avoid either of the saturation regions. In the saturating regions, the derivative of the neuron becomes 0 or near zero, thus aggravating the problem of vanishing gradients [81]. We used dropout (set to 0.4) [38] during training for full precision networks with ReLU activation function. In case of networks with binary activations, no dropout was applied, as the binary neurons themselves worked as a regularizer similar to dropout. The output of the final layer was fed to a softmax layer and the negative loss likelihood on the softmax function was defined as the network loss. We trained the networks for 150 epochs while keeping the batch-size at 50, and we used Adam optimizer[37] for weight update. The learning rate was set to 0.01 initially and was halved every 25 epochs. The highest test accuracy observed during the training epochs is reported at the end in Table 4.3.

4.2.3 Experiments with CIFAR-100

For CIFAR-100, we used the 16 layer VGG-16 [61] architecture composed of 3×3 convolutional filters. The network architecture is outlined in Table 4.2(b). We trained the network for 3 different weight quantization schemes and 5 different neuron activation functions, as described in Sec. 4.1. At every layer, except the last layer, batch-normalization [65] is applied. We used dropout (set to 0.5) [38] during training for full precision networks with ReLU activation function. For networks with binary activations no dropout was applied. The output of the final layer was fed to a softmax layer and negative loss likelihood was used as the loss function. For all the cases, we trained the network for 400 epochs, starting with a learning rate of 5e-3 which was halved every 100 epochs, while keeping the batch-size at 50.

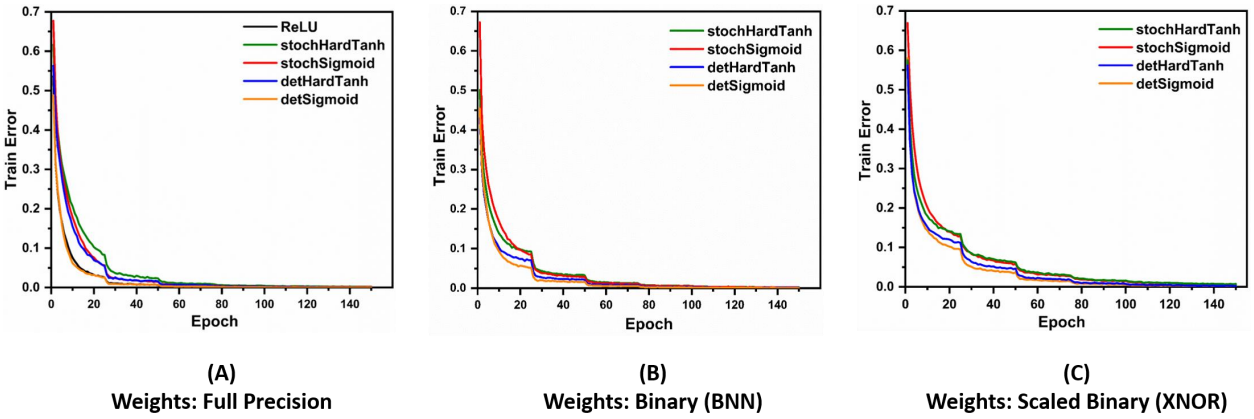


Figure 4.1. : Training Error over epochs for 5 different activations and 3 different weight quantizations for CIFAR-10 on VGG-9

Adam [37] was the optimization function used. The highest test accuracy observed during the training epochs is reported at the end in Table 4.4.

4.3 Results

4.3.1 CIFAR-10

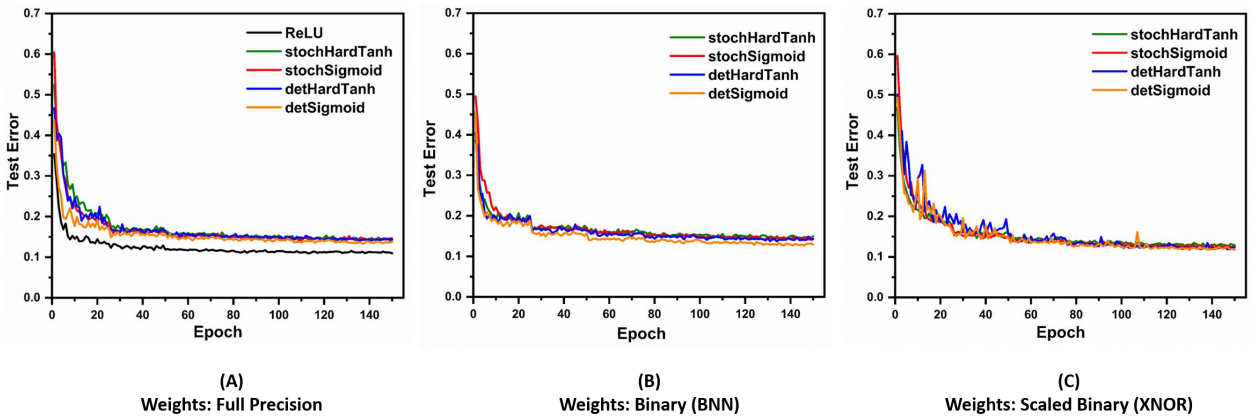


Figure 4.2. : Test Error over epochs for 5 different activations and 3 different weight quantizations for CIFAR-10 on VGG-9

For CIFAR-10, we achieve training convergence for all 4 types of binary activations (Fig. 4.1). The final test accuracy of the SNNs was at par with the ANN with ReLU (89.07%), recording the lowest degradation of 0.8% in accuracy for *detSigmoid* with XNOR-Net based

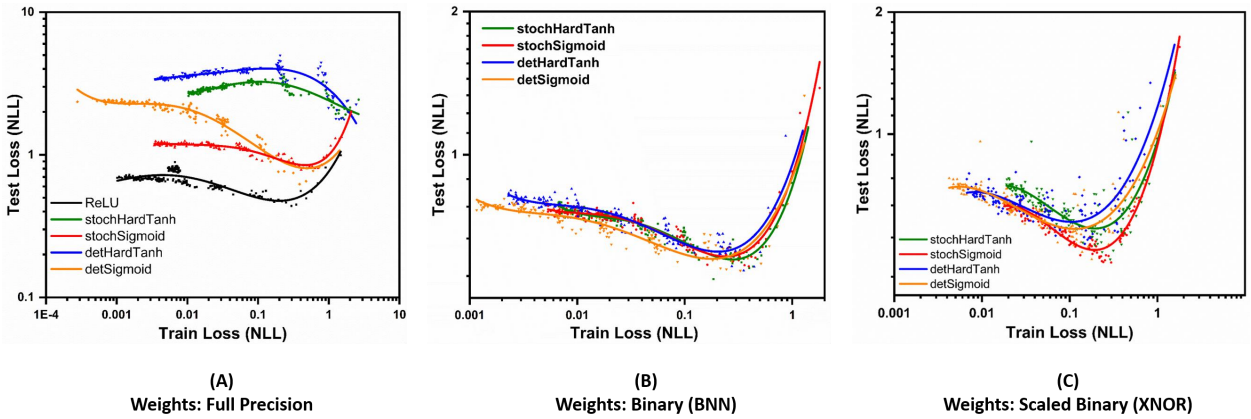


Figure 4.3. : Test Loss vs Train Loss Trajectory of the VGG-9 for CIFAR-10. The lines are fourth degree polynomial fit of the data (the plotted points)

Table 4.3. : Test Accuracy CIFAR-10 VGG-9

	Weights		
Activation	Full Precision	BinaryNet	XNOR-Net
ReLU	89.07%	N/A	N/A
stochHardtanh	86.13%	85.69%	87.64%
stochSigmoid	86.08%	85.95%	87.95%
detHardtanh	85.88%	86.30%	88.16%
detSigmoid	86.68%	87.37%	88.27%

quantized weights and a maximum degradation of 3.38% for *stochHardtanh* with BinaryNet based quantized weights (Table 4.3, Fig. 4.2). An interesting observation was that XNOR-Net based quantized networks performed better than full precision networks with binary activations. A possible reason is that in XNOR-Nets keep ReLU activation for the first and last layers of the networks a design requirement. However, in the full precision networks, we converted activations of all the layers to binary.

We also analyzed the negative log likelihood loss (NLL) of these networks for both test and train data to better understand the generalization ability of these networks (Fig. 4.6). Ideally we want the test loss to reduce as training loss reduces. However, if test loss starts to increase with decreasing train loss, it is indicative of overfitting. In full precision networks, ReLU activation gave us the best generalization. Among binary activations, both *stochSigmoid*

and *detSigmoid* performed much better than their binary hardtanh counterparts. We observe similar trends even in BinaryNet and XNOR-Net version of VGG9, where the binary sigmoid neurons exhibit better generalization than binary hardtanh neurons.

When we compare the loss of the neural networks (Fig. 4.6), we observe that *stochSigmoid* and *stochHardtanh* have a more downward curve compared to their deterministic counterparts, thereby indicating better generalization. In term of test accuracy, from Table 4.3, one may observe that deterministic binary neurons performed better than their stochastic counterparts for quantized SNNs, barring one case of full precision SNN with *stochSigmoid* neurons. However, the difference is less than 1% between the deterministic and stochastic binary neurons. In the next section, we evaluate these activations for a more complex task on an almost twice as deep neural network.

4.3.2 CIFAR-100

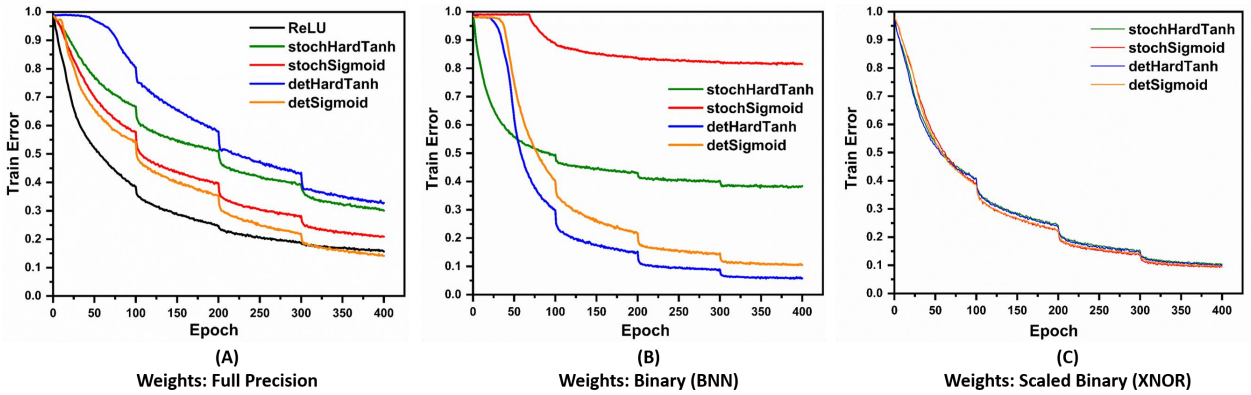


Figure 4.4. : Training Error over epochs for 5 different activations and 3 different weight quantizations for CIFAR-100 on VGG-16

For VGG-16 neural network architecture, the baseline accuracy is 61.39% for full precision ANN with ReLU neurons. When training SNNs, there was significant reduction in the test accuracy due to binary activations (Table 4.4). Also, we observed that to train these full precision SNNs it was necessary to clip and clamp the 32-bit weights within 1,-1 range.

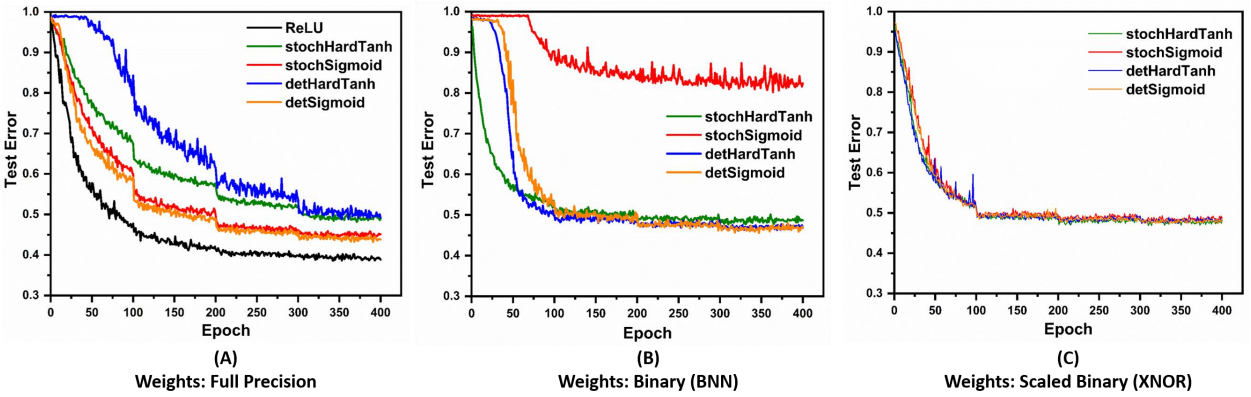


Figure 4.5. : Test Error over epochs for 5 different activations and 3 different weight quantizations for CIFAR-100 on VGG-16

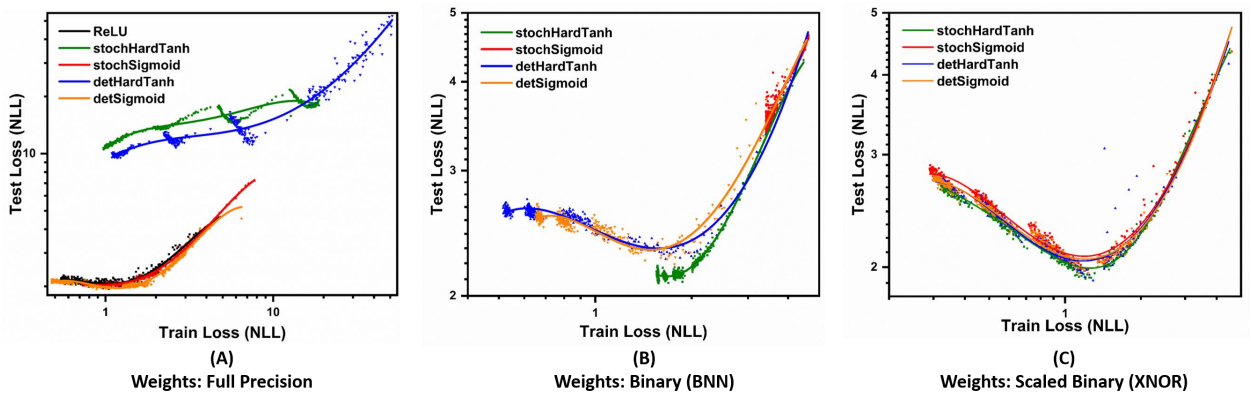


Figure 4.6. : Test Loss vs Train Loss Trajectory of the VGG-16 for CIFAR-100. The lines are fourth degree polynomial fit of the data (the plotted points)

In these SNNs binary sigmoid neurons performed significantly better than binary hardtanh neurons, by almost a margin of 5%, as noted in Table 4.4.

For BinaryNet based weight quantization, the SNN with *stochSigmoid* neurons failed to train properly (Fig. 4.4B, 4.5B). The SNN with *stochHardtanh* neuron also suffered from poor training convergence. Thus one can conclude that stochasticity is not ideal for such binarized SNNs.

On the other hand, for SNNs with XNOR-Net based weight quantizations, the train and test error trajectory was similar for all 4 kinds of binary activations. Following the trend observed in Sec. 4.3.1, quantized SNNs perform at par or even better than full precision SNNs.

Table 4.4. : Test Accuracy CIFAR-100 VGG-16

	Weights		
Activation	Full Precision	BinaryNet	XNOR-Net
ReLU	61.39%	N/A	N/A
stochHardtanh	51.82%	52.36%	53.11
stochSigmoid	55.95%	19.81%	52.54
detHardtanh	51.2%	53.83%	52.71%
detSigmoid	56.78%	54.44%	52.64%

Thus it can be concluded that for deeper SNNs, networks with XNOR-Net based weight quantization are preferred over full-precision and BinaryNet based quantized networks.

We also analyzed test loss versus train loss for VGG-16 as presented in Fig. 4.3. For full precision weights, binary sigmoid neurons performed as well as ReLU neurons in terms of loss convergence and generalization. However, for quantized SNNs, the binary hardtanh neurons offered better performance. In XNOR-Net based weight quantized SNN, *stochHardtanh* neuron achieved the highest accuracy of 53.11%. And its test loss vs train loss curve is the lowest of all 4 in Fig. 4.3C.

For BinaryNet based weight quantization, the SNN with *stochHardtanh* activations has the lowest test loss vs train loss curve (Fig. 4.3). However, it doesn't translate to best performance in this case. While the difference between train and test loss is usually a good indicator of generalization, it is important to note that such comparisons can be done only when train error is very low. However, in this case, train error of *stochHardtanh* is very high, as seen in Fig. 4.4B. Hence, the next candidate for best performance and generalization is the orange curve of *detSigmoid* neurons, and it does achieve the highest test accuracy of 54.44%.

For XNOR-Net based weight quantization, *stochHardtanh* achieves the highest test accuracy of 53.11%. From Table 4.4, one should note that deterministic binary neurons performed better than their stochastic counterparts for most cases, except for binary *hardtanh* neurons in XNOR-Net based weight quantized SNN, where the stochastic *hardtanh* neuron outperformed its deterministic counterpart by 0.4%.

4.4 Hardware Implications

We have discussed earlier how binary stochastic activations promise efficiency in terms of energy consumption in hardware implementations. To assess the energy benefits, we analyzed the energy consumption for neuronal activity of networks with binary stochastic activations. For comparison, we consider networks with full-precision activations to show the energy benefits obtained by using binary activations. Note, along with energy improvements at a neuronal level, binary activations also eliminate the need for the costly multiply-and-accumulate (MAC) operations, as the operations are simplified to additions and subtractions of weights. We calculated the energy consumption using estimates for 45 nm CMOS technology [82], [83]. Considering 32-bit representation as full-precision, the energy consumption for both 32-bit memory accesses and computations are shown in Table. 4.5. Here, p is the bit-precision of the access. The estimated energy consumption of different networks is listed in Table. 4.6

Table 4.5. : Energy Consumption chart

Operation	Term	Energy (pJ)
p-b Memory Access	E_{A-p}	$2.5p$
32-b MULT FP	E_{M-F}	3.7
32-b ADD FP	E_{AD-F}	0.9

Table 4.6. : Energy estimates for different networks

	Energy (mJ/inference)	
Weights	Activations	
	Full Precision	Binary
Full Precision	4.287	3.02 (1.4x)
BinaryNet	–	0.152 (28.1x)
XNOR-Net	–	0.2 (21.3x)

We observe that in a network with full-precision weights, binary activations reduces the energy consumption by $\sim 40\%$. Moreover, networks with binary weights achieve more than 21x improvement in energy consumption over full-precision networks.

4.5 Conclusion

In this work, we explored the scalability of deep probabilistic spiking neural networks. We observe that binary neurons are able to preserve the performance in deep networks, thus paving the way for larger applications of such SNNs. The spiking neural networks (SNNs) with stochastic activations performed at par with deterministic SNNs, thus opening avenues for stochastic neuro-mimetic hardware platforms. We also observed that unipolar neurons performed at par with bipolar neurons (except for one case) which allows for further optimization as unipolar neurons have the advantage of inherent sparsity. This sparsity can be exploited in hardware design to improve the energy efficiency. Finally, we performed an energy consumption analysis to show the benefits of using binary activations over full-precision counterparts for networks with both full-precision and binarized weights.

5. INTRINSIC ADVERSARIAL ROBUSTNESS OF ANALOG COMPUTING

5.1 Introduction

To accommodate the growing computational needs of DNNs special-purpose accelerators such as GoogleTPU [84], Microsoft BrainWave [85], and NVIDIA V100 [86] have been proposed. These systems operate on the principle of efficiently performing matrix-vector multiplication (MVM) operations, the key computational kernel in DNNs, by co-locating memory and processing elements. Despite their success, the saturating scaling trends of digital CMOS [87] has garnered interest in non-volatile memory (NVM) technologies such as RRAM [9], PCRAM [10] and Spintronics [88]. The memory element in these technologies can be arranged in a crossbar fashion to enable efficient MVM computations in the analog domain inside the memory array. Such an in-memory computing primitive can significantly lower power and latency compared to digital CMOS [8]. Promises offered by the NVM crossbars have propelled significant research in designing analog computing based accelerators, such as PUMA [89].

In an analog computing hardware the output of an MVM operation is sensed as a summation of currents through resistive NVM devices arranged in a crossbar, and hence are prone to errors due to non-ideal behavior of the crossbar and its peripheral circuits. Such errors are hard to model due to the interdependence of multiple analog variables (voltages, currents, resistances) in the crossbar. These deviations result in overall performance degradation of the DNN implementation [90]. Several works have explored various techniques to counteract the impact of these non-idealities [91], [92].

On the flip side, even though the changes in DNN activations arising from non-idealities is hard to model, it can potentially lead to adversarially robust DNN implementations. Adversarial images are generated by estimating the gradients of the model with respect to its input, and carefully perturbing the images in the direction of maximum change in the classifier output [14], [93]. To counter such attacks, several techniques that rely on gradient obfuscation have been previously proposed [94]–[96]. In this chapter, we explore how non-ideal NVM crossbars have a similar intrinsic effect of gradient obfuscation. We

implement DNNs on the PUMA architecture, which is composed of thousands of MVM units (MVMUs) made of NVM crossbars. The aformentioned errors occur at the output of these internal MVMUs, which are practically inaccessible to a third party user, such as the software designer or even an attacker. Moreover, the nature of the errors depends heavily on the technology, which might not be fully disclosed by the manufacturer. Finally, any scaled technology is prone to chip to chip variations [97] which can further deter an attacker from exactly replicating the DNN activations. We study two distinct scenarios, one where the attacker does not have access to custom NVM hardware and generates attacks based on accurate digital hardware, and the other where the attacker generates attacks with the NVM hardware in loop.

The main contributions of the work presented in this chapter are as follows:

- We demonstrate that adversarial attacks crafted without the knowledge of the hardware implementation are less effective in both black box and white box scenarios.
- We tested multiple variants of NVM crossbars, and show that the degree of intrinsic robustness offered by the analog hardware is in proportion to its degree of non-ideal behavior.
- We show that "Hardware-in-Loop" adaptive adversarial attacks are more effective, as the attacker can now account for the non-ideal computations when crafting the adversarial examples. We show that the degree of success depends on what hardware is available to the attacker and how similar it is to the target model's hardware.

5.2 Background and Related Work

5.2.1 In-memory Analog Computing Hardware

In-memory analog computing with NVM technologies are being extensively studied for machine learning (ML) workloads [98]–[100] because of their inherent ability to perform efficient matrix-vector multiplications, the key computational kernel in DNNs. The basic compute fabric in NVM technologies is a two-dimensional cross-point memory, known as a crossbar, shown in Fig. 5.1. The memory devices lie at the intersection of horizontally (source-line) and vertically (bit-line) running metal lines. The conductance of each memory

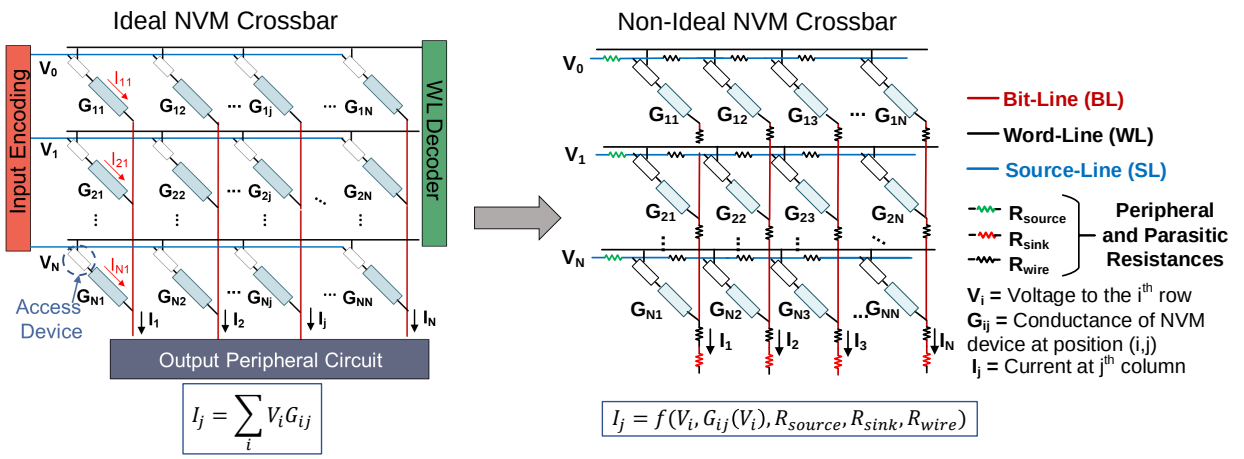


Figure 5.1. : (Left) Illustration of NVM crossbar which produces output current I_j, as a dot-product of voltage vector, V_i and NVM device conductance, G_{ij}. (Right) Various peripheral and parasitic resistances modify the dot-product computations into an interdependent function of the analog variables (voltage, conductance and resistances) in a non-ideal NVM crossbar.

device can be programmed to a discrete number of levels [101]. By simultaneously applying inputs, in the form of voltages, V_i, at the source-lines, the multiplications are performed between the voltages, V_i and conductances, G_{ij}, by each NVM device using the principle of Ohm's law. Finally, the product, which is the resulting current, I_{ij}, from each NVM device, is summed up using Kirchoff's current law to produce a dot-product output, I_j at each column:

$$I_j = \sum_i I_{ij} = \sum_i V_i G_{ij} \tag{5.1}$$

Such parallel dot-products across all columns enable efficient multiplication of the input voltage vector, V, and the crossbar conductance matrix, G, resulting in an output vector, $I = VG$. A few key aspects of the design of NVM crossbars are the following parameters:

- Crossbar Size: The number of rows and columns in the crossbar matrix.
- ON Resistance (R_{ON}): The minimum resistance level of the NVM device.

Typically, in a convolutional neural network (CNN), the convolution operation between the input and the weight tensor can be represented in the form of a series of MVM operations, which can be subdivided into smaller MVM operations to conform to the technological restrictions of the size of the NVM crossbar. Floating point inputs and weights in DNNs

are converted to fixed point precision to make them compatible with NVM crossbar based computations.

The analog nature of computing in NVM crossbars introduces functional errors in the MVM computations due to several non-idealities arising from the NVM devices and peripheral resistances. The aforementioned crossbar design parameters, such as Crossbar Size, and ON Resistance have varying impact on the degree of functional errors introduced by the non-idealities [90] by affecting the effective resistance of a crossbar column. Larger crossbar size lowers the effective resistance, making the crossbar more prone to non-ideal effects, while higher ON resistance increases it, resulting in a crossbar less affected by non-idealities.

Due to the non-idealities, the resulting output current, I_{ni} is a function of volatge vector V, conductance matrix $G(V)$, which is now dependent on V, and several non-ideal factors:

$$I_{ni} = f(V, G(V), R_{source}, R_{sink}, R_{wire})5.2 \tag{5.2}$$

To study the impact of such non-ideal behavior of NVM crossbars on DNNs, researchers have previously proposed techniques to model the non-ideal function in Equation 5.2. One such technique is GENIEx [90] where the authors use deep learning to model the aforementioned non-ideal function.

Modeling of Non-Idealities using GENIEx

The understand the behavior of NVM crossbars in presence of non-idealities, it is necessary to accurately model Equation 5.2. Since I_{ni} is a function of voltage, V and G, and G is a function of voltage, Equation 5.2 needs to be solved in a self-consistent manner instead of solving a set of linear equations. Thus, the NVM device non-linearity adds a new dimension to crossbar modelling. We use GENIEx crossbar modeling technique [90], which uses a 2 layer perceptron network to model Equation 5.2, that is, to predict a function of the output current, $f(I_{ni})$, from the crossbar for different input voltages, V and conductances, G of the NVM crossbar. More specifically, the network predicts the ratio: $\delta_{ni} = I_{ni}/I_{ideal}$ since $I_{ideal} = V * G$ is already known. In order to train the GENIEx perceptron network, we obtain data in form of training pairs: $[(V, G),\ f(I_{ni}) = \delta]$ by performing circuit simulations

on NVM crossbar for different V and G combinations. Since the training data is obtained directly from circuit simulations on NVM crossbar, it depends on the property of the NVM crossbar, specifically, its size, ON resistance as well as bit representation of the NVM device and input voltages. Thus, each time one of these properties of a crossbar varies, it would lead to a different GENIEx model.

After obtaining data from the circuit simulations, we normalize the data to create a dataset, and train a 2 layer perceptron network to obtain the GENIEx crossbar model. The crossbar model has been demonstrated to achieve outputs close to circuit simulations on test V and G data [90].

Functional Simulator for Crossbar Architectures: PUMA

In order to evaluate deep learning models on NVM crossbars in presence of non-idealities, we use a simulation framework [90] following the standard technique of mapping convolutional and linear layers on a spatial NVM crossbar architecture such as PUMA [89]. This mapping is composed of three parts: i) Iterative matrix-vector multiplications, ii) Tiling and iii) Bit-slicing. First, the convolution or linear layer operation in a neural network, is divided into iterative matrix vector computation. For example, a typical convolutional layer with kernel-size $k \times k$, I input channels and O output channels, is represented as a matrix $k^2I \times O$ and each stride of the input tensor is flattened into a vector of size k^2I. This matrix-vector multiplication operation is performed iteratively for each stride. Second, since crossbar size is limited, the weight matrix is tiled into a number of crossbar sized segments. Third, since NVM devices can only accommodate limited number of bits, to represent larger bit precision inputs and weights, we perform bit-slicing. Here inputs and weights are divided into smaller bit segments, and each segment is mapped on individual crossbars. Thus, each crossbar is characterized by its size, device properties and bit width of each segment of inputs and weights. Based on these characteristics, the crossbar model is obtained using the aforementioned GENIEx technique. The results of computations performed by each crossbar is obtained by applying bit-sliced inputs and weights to the GENIEx crossbar model. This simulation ecosystem accurately captures the effect of various non-idealities on classification

performance of neural networks when mapped on NVM crossbar hardware. DNNs typically consist of thousands of MVM operations at every layer. The NVM crossbar non-idealities cause the activations at every layer to deviate from their expected value, and this deviation propagates through the network. This results in a degradation of DNN accuracy at inference (without any adversary). Interestingly, the same deviation in activation imparts adversarial robustness when under attack, which is further analyzed in this work.

5.2.2 Adversarial Attacks

In 2013, the authors of [14] demonstrated that a classifier can be forced to make an error by adding small perturbations to the data which are almost imperceptible to the human eye. They coined the term "adversarial examples" to define such data designed specifically to fool the classifier. Since then, several methods have been developed to generate such data, which are known as "adversarial attacks". In principle, these attacks try to solve the following optimization problem [102]:

$$x^* = x + argmin\{z : F(\theta, x+z) \neq F(\theta, x)\} = x + \delta_x \tag{5.3}$$

where x is the original data, x^* is the perturbed adversarial data, θ is the model parameter, $F(\theta, x)$ is the classifier function, mapping inputs to labels, and the objective of the adversary is to misclassify, i.e. $F(\theta, x^*) \neq F(\theta, x)$. Most attacks use gradient-based optimization to solve for eq.5.3, and the attack's success relies on how accurately one can estimate $\nabla_x L(\theta, x, y)$, the derivative of the cost function $L(\theta, x, y)$ with respect to x [93].

5.3 Adversarial Robustness of NVM Crossbar based Analog Computing

In recent years, several adversarial defenses have been proposed that disrupt the gradient computation of the model by adding an extra computational element to the network, such as a randomization layer at the beginning [103], or adaptive dropout after every layer [94]. When a DNN model is implemented on an NVM crossbar architecture, the non-idealities have a similar effect of changing the layer-wise activations of the DNNs. There is no simple

differentiable function to model these deviations, and one cannot determine them without probing the analog hardware. Thus, such an implementation, could potentially increase the robustness of the neural network. In this section we describe the methodology to emulate DNNs on the PUMA architecture, and set up different threat models based on the attacker's knowledge of both the software and the hardware.

5.3.1 Crossbar Models

Table 5.1. : Crossbar Model Description

	Crossbar parameters		
Crossbar Model	Size	R_{ON} (Ω)	NF
64×64_300k	64×64	300k	0.07
32×32_100k	32×32	100k	0.14
64×64_100k	64×64	100k	0.26

To model the non-ideal crossbar, we use GENIEx, a deep learning based crossbar model developed by the authors of [90]. They define a multi-layer perceptron (MLP) which receives V and G as inputs and predicts the output I_{ni}. This MLP is trained on training pairs [(V,G), I_{ni}] obtained from circuit simulations. In this work, we have replicated the modeling technique of GENIEx to generate 3 crossbar models (Table 5.1). For this work, we have used the RRAM device model as the NVM device in the crossbar.

The degree of non-ideality has been described by the authors of GENIEx as Non-ideality Factor (NF) = (Expected output-Actual Output)/Expected Output. NF is directly (inversely) proportional to crossbar size (ON Resistance). In our experiments, we have considered different crossbar models to study the impact of different degrees of non-idealities, represented by different NF, on adversarial robustness, as shown in Table 5.1. For example, a high R_{ON} results in higher effective resistance in the column, resulting in relatively higher voltage drop across the device compared to the voltage drop in the parasitic and peripheral non-ideal resistances. Thus, a crossbar with higher R_{ON} has lower NF. On the other hand, a higher crossbar size consists of more resistance in parallel in a column of a crossbar, resulting in lower effective resistance, which causes higher NF. Thus different crossbar types

have significant impact on NF, and in this work, we study the adversarial robustness of each of these crossbars. To implement this, we train different GENIEx crossbar models by creating datasets from data obtained by performing circuit simulations on the crossbar types listed in Table 5.1. To integrate these NVM crossbar models with the PyTorch framework, we have adopted the aforementioned PUMA functional simulator from [90] based on PUMA hardware architecture [89].

5.3.2 Datasets and Network Models

For our evaluation we selected 3 image recognition tasks, and trained a ResNet [67] for each task.

- **CIFAR-10** [60]: A ResNet-20 was trained for 200 epochs, with the learning rate (lr) schedule $[0.1(1, 79), 0.01(80, 119), 0.001(120, 200)]$ and achieved test accuracy of 92.44%
- **CIFAR-100** [60]: A ResNet-32 was trained for 200 epochs, with the lr schedule $[0.1(1, 79), 0.01(80, 119), 0.001(120, 200)]$, and achieved test accuracy of 71.42%
- **ImageNet** [104]: A ResNet-18 was trained for 90 epochs, with the lr schedule $[0.1(1, 29), 0.01(30, 59)$, and achieved top-1 and top-5 test accuracy of 69.83% and 89.19% respectively. We used a reduced test set of 1000 images for adversarial attacks.

5.3.3 Generating Adversarial attacks

Table 5.2. : Attacker's Knowledge for the Threat Scenarios

		Accurate Digital Computation		Non-Ideal Analog Computation		
Attack Type	Model Weights	Logits	Activations	Crossbar Model	Logits	Activations
Non-Adaptive Attacks						
Transfer Attacks	No	No	No	No	No	No
Black Box Attacks	No	Yes	No	No	No	No
White Box Attacks	Yes	Yes	Yes	No	No	No
Adaptive Attacks						
Black Box Attacks	No	N/A	N/A	Yes (may not match)	Yes	No
White Box Attacks	Yes	N/A	N/A	Yes (may not match)	Yes	Yes

We define 5 different threat scenarios with varying extent of the attacker's knowledge of the target model and the underlying hardware (Table 5.2) and created gradient based

attacks. For each scenario, we defined an attack model (a single DNN or an ensemble of DNNs) to generate the adversarial images. We use Projected Gradient Descent (PGD) [105] to generate iterative perturbations that are bound by the l_∞ norm, as shown in Eq.5.4:

$$x^{t+1} = \Pi_{x+S}(x^t + \alpha sgn(\nabla_x L(\theta, x^t, y)) \tag{5.4}$$

x^{t+1} is the adversarial example generated at $(t+1)^{th}$ iteration. The model's cost function is $L(\theta, x, y)$, which is a function of the model parameters θ, input x, and labels y. The set of allowed perturbations is given by S. For the l_∞ norm, the attack epsilon (ϵ) defines the set of perturbations as $S = \Big(\delta | \Big(x + \delta \geq max(x + \epsilon, 0)\Big) \wedge \Big(x + \delta \leq min(x + \epsilon, 1)\Big)\Big)$, where $x \in [0, 1]$.

Additionally, for the two threat scenarios, non-adapative, and adaptive black box attacks we also generated adversarial images using Square Attack [106], which is a query efficient adversarial black box attack. While PGD attack success is dependent on estimating the local gradients of the defending model, such a query based attack doesn't rely on gradient information at all. Instead, it generates adversarial images by conducting a randomized search [107], [108]. Every time the attacker queries the model, the input image has random perturbations, sampled from a given distribution. If the perturbation succeeds in increasing the loss for that image, the image is updated, and this continues till either the image is misclassified, or the query limit is reached.

Non-Adaptive Attacks

Our first category of threat scenario is "Non-Adaptive Attacks", i.e. the attacker has no knowledge of the underlying analog hardware and the attacks are generated under the assumption of accurate digital computation. Under this category, we have 3 varying degrees of attack.

Transfer Attacks: This is the weakest threat model where the adversary has no knowledge of the model. The attack model is another DNN trained on the same dataset and run on an accurate digital hardware. The attack model architectures for CIFAR-10/100, and Imagenet are ResNet-10, ResNet-20, and AlexNet [2], respectively. ResNet-10 and ResNet-20

are trained on CIFAR-10/100, respectively, using the same training schedule as the target models. For Imagenet, we used a pretrained AlexNet available in Pytorch [109].

Ensemble Black Box Attacks: The attacker queries the model on an accurate digital hardware and reads the output of the final layer before softmax (logits) to generate a synthetic dataset of training data and its corresponding logits. This synthetic dataset is used to train 3 different surrogate ResNet models, ResNet-10,20,32. These 3 models are then used to generate adversarial images using the stack parallel ensemble strategy [110].

Square Attack (Black Box) The attacker queries the model on accurate digital hardware and has access to the last layer before softmax (logits) as in the case of the Ensemble Black Box Attacks. We use l_∞ Square Attack [106], and set the maximum query limit to 1000.

White Box Attacks This is the highest threat level where the attacker has full knowledge of the model weight, thus the attack model is the same as the target model. However, while generating gradients, the attacker has no knowledge of the underlying analog hardware implementation. The gradients for the attack are computed assuming accurate digital hardware implementation.

Hardware in Loop Adaptive Attacks

In this category of attacks, the attacker is aware that the model is implemented on an NVM crossbar hardware. However, the crossbar model available to the attacker may or may not match with the target's implementation.

Ensemble Black Box Attacks: For training the surrogate models, the attacker queries the DNN model implemented on the NVM crossbar based hardware to create the synthetic dataset. We use 3 different crossbar models as defined in Table 5.1 and we explore scenarios where there is a mismatch in the crossbar model used by the attacker and the target implementation. We selected 64x64_100k as the NVM crossbar model available to the attacker.

Square Attack (Black Box): The attacker runs the iterative query based Square attack on the DNN implemented on the NVM crossbar based hardware. As emulation of the crossbar based architecture take much longer, we limit the total number of queries to 30.

We selected 32x32_100k as the NVM crossbar model available to the attacker, and used all 3 crossbar models for defense.

White Box Attacks: In the case of White Box attacks, the attacker generates adversarial images using "Hardware-in-Loop" gradient descent. Note that the NVM crossbar based hardware is designed for inference tasks and does not support backpropagation of gradients. Thus, for "Hardware-in-Loop", the forward pass is performed on NVM crossbar hardware, and all activations are recorded. However, the derivatives are calculated assuming ideal computations in place of non-ideal MVM operations of the crossbar. As described in Section 5.3.1, the NVM crossbar non-idealities vary with crossbar properties. We selected 64x64_100k as the NVM crossbar model available to the attacker, and used all 3 crossbar models for defense.

Comparison with Related Work

We have selected 3 defenses that can be applied to a pretrained network as listed below. For a fair comparison, we apply non-adaptive attacks for these defenses as well, i.e. the defenses are not visible to the attacker when they query the model to generate their synthetic dataset for Black Box attacks, and when they generate gradients for White Box attacks.

- **Input Bit Width (BW) Reduction** [111]: The input is quantized to 4-bits.
- **Stochastic Activation Pruning (SAP)** [94] (for CIFAR-10/100 only): At inference, after every convolution layer, there is an adaptive dropout, that randomly sets the layer outputs to 0 with a probability proportional to their absolute value.
- **Random Padding** [103] (for ImageNet only): Two randomization layers are introduced before the pretrained model. The first layer scales the input image to a random size NxN where N $\in$ [299, 331] using nearest-neighbor extrapolation. The second layer randomly pads the image to generate the final image of size 331x331.

5.4 Results

The first effect of implementing DNNs on a NVM crossbar hardware is the reduction in clean accuracy due to the errors associated with non-ideal computations. Greater the Non-

Ideality Factor (NF), more severe is the accuracy degradation as noted in Table 5.3. The clean accuracy of CIFAR-10 drops from 92.44% (accurate digital hardware) to 88.34% on 64x64_100k, the most non-ideal crossbar model among the three chosen. Similarly, CIFAR-100 accuracy drops from 71.42% to 55.48% and ImageNet accuracy falls from 69.56% to 62.50% on the 64x64_100k NVM crossbar hardware. If non-idealities of NVM hardware had no impact on adversarial robustness, similar degradation would have been observed in model accuracy under attack. However, our findings, as outlined below, indicate a different trend.

5.4.1 Non-Adaptive Attacks

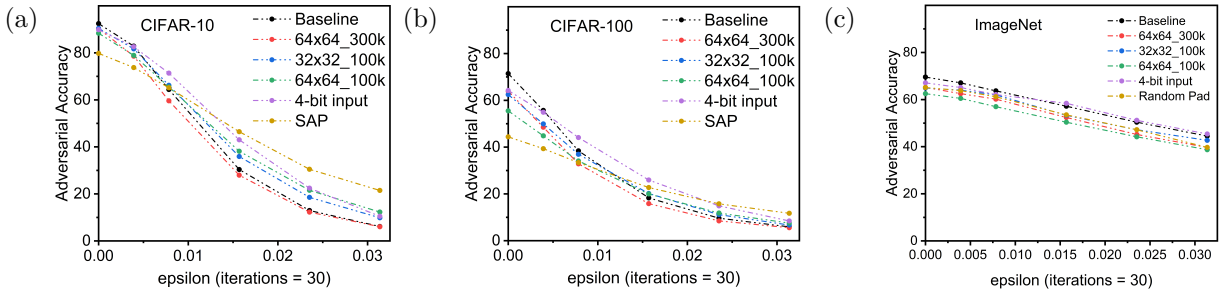


Figure 5.2. : Non-Adaptive Transfer Attacks (PGD, iter=30) on CIFAR-10/100 and ImageNet on 3 NVM models and 3 defenses, Input BW Reduction (4-bit input) [111], SAP [94], Random Pad [103]

Transfer Attacks: In Fig. 5.2, we observe the decline in adversarial accuracy with increasing attack epsilon (ϵ) for CIFAR-10, CIFAR-100, and Imagenet. For CIFAR-10/100, the 64x64_300k model did not exhibit any increase in robustness, instead it trailed behind the baseline accuracy. In case of CIFAR-10, the other two crossbar models, 32x32_100k and 64x64_100k, displayed an absolute increase in robustness of 4.2% and 5.9% averaged over ϵ = (2,4,6,8)/255, respectively. For CIFAR-100, the average increase in robustness for ϵ = (4,6,8)/255 was 1.4% for 32x32_100k and 1.84% for 64x64_100k. The peak improvement in robustness was observed for $\epsilon = 6/255$ and has been summarized in Table 5.3. For ImageNet, we do not observe any improvement in robustness (Fig. 5.2c(c)). A possible reason could be that the attack is much weaker, as it was generated on a different architecture (AlexNet),

instead of a ResNet. The more generic the attack, the less effect the NVM non-idealities seem to have on robustness.

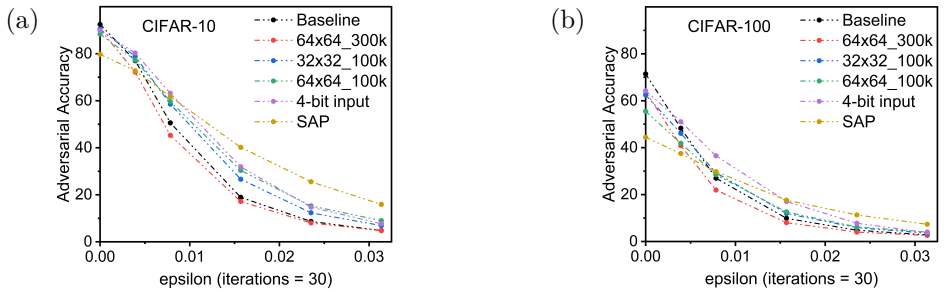


Figure 5.3. : Non-Adaptive Ensemble (Black Box) PGD (iter=30) on CIFAR-10, CIFAR-100 on 3 NVM crossbar models and the 2 defenses, Input BW Reduction (4-bit input) [111] and SAP [94]

Ensemble Black Box Attacks: From Fig. 5.3, we observe similar trends as transfer attacks for CIFAR-10, and CIFAR-100. The 64x64_300k model didn't exhibit any increase in robustness, instead it trailed behind the baseline accuracy. The NVM crossbar models, 32x32_100k and 64x64_100k, recorded an absolute increase in robustness of 5.3% and 7.8% averaged over ϵ = (2,4,6,8)/255, respectively for CIFAR-10. For CIFAR-100, it was 1.4% and 1.84% respectively. The peak improvement in robustness was observed for ϵ = 4/255 and has been summarized in Table 5.3.

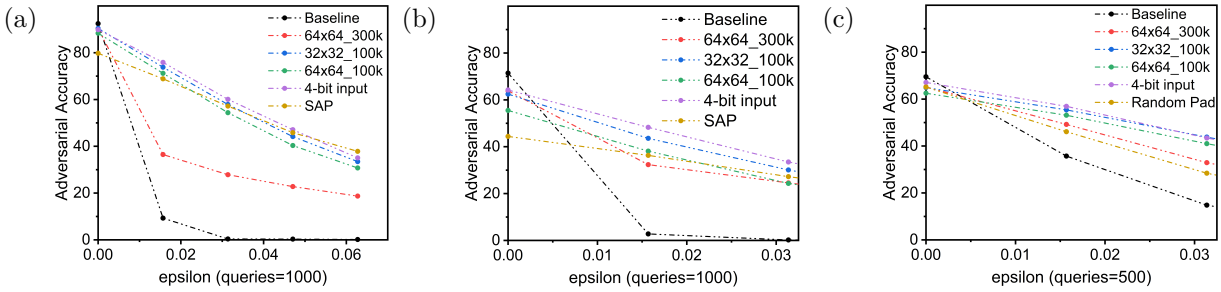


Figure 5.4. : Non-Adaptive Square Attacks (Black Box) on CIFAR-10/100 and ImageNet on 3 NVM models and 3 defenses, Input BW Reduction (4-bit input) [111], SAP [94], Random Pad [103]

Square Attack (Black Box): The analog hardware shows the highest resilience against such an attack. As the attack is gradient-free in nature, we conclude that analog hardware offers robustness by modifying the inference itself. The perturbations that cause complete

model failure, i.e. 0% accuracy, have much lower impact on the model implemented on NVM crossbar. The other 3 defense methods [94], [103], [111], also perform well over a wide range of ϵ=(4,8,12,16)/255. The average robustness gain observed for CIFAR-10 was 23.93% , 49.80% and 46.63% with crossbar models as 64x64_300k, 32x32_100k, 64x64_100k respectively. We see robustness gain increase from 64x64_300k to 32x32_100k, and then drop slightly for 64x64_100k. The increase if due to higher deviations in 32x32_100k compared to 64x64_300k. The slight decrease however, can be attributed to the counter effect of inaccurate computations as non-idealities increase further. We observe similar trends in CIFAR-100 and Imagenet as well, as shown in Fig. 5.4 and Table 5.3.

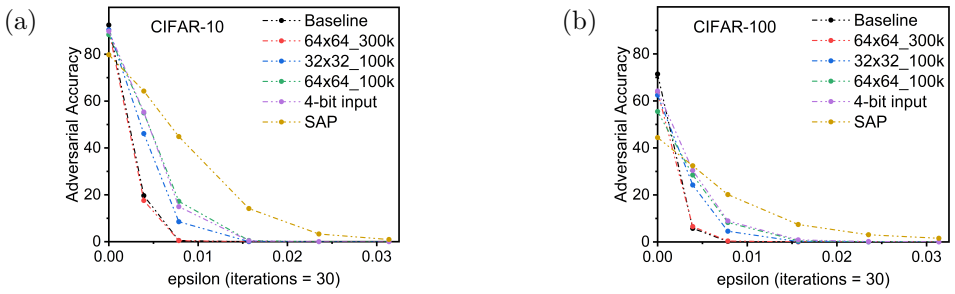


Figure 5.5. : Non-Adaptive White Box Attacks (PGD, iter=30) on CIFAR-10, CIFAR-100 on 3 NVM models and 2 defenses, Input BW Reduction (4-bit input) [111] and SAP [94]

White Box Attacks: Under this threat model we observe much improvement in robustness as depicted in Fig. 5.5 and Table 5.3. The NVM model 64x64_300k still continues to closely follow baseline accuracy. For all 3 datasets, the baseline accuracy drops sharply to 0 beyond $\epsilon = 2/255$. At this level, the NVM models are no longer able to recover any performance. For ϵ = (1,2)/255, we observe that 64x64_100k, the most non-ideal of the 3 models, offers the highest improvement for all 3 datasets, with absolute increase of 35.34% for CIFAR-10, 22.69% for CIFAR-100, and 9.90% for ImageNet at $\epsilon = 1/255$.

We have summraized below the common trends observed across all 5 non-adaptive attacks.

- For gradient based attacks (PGD), more the attacker relies on estimating the true gradinets of the target model for attack generation, greater is absolute robustness

Table 5.3. : Summary of Non-Adaptive Attacks on NVM Crossbar Models

Attack Type	Baseline	NVM Crossbar Models (Target) 64×64_300k	32×32_100k	64×64_100k	4-bit input [111]	SAP [94]
		CIFAR-10 (ResNet-20) (test samples = 10000)				
Clean	92.44	90.35 (-2.09)	90.42 (+2.02)	88.34 (-4.10)	89.84 (-2.60)	79.76 (-12.68)
Transfer (ResNet-10) PGD ϵ = 6/255, iter = 30	12.94	12.24 (-0.70)	18.53 (+5.59)	21.54 (+8.6)	22.43 (+9.49)	30.48 (+17.54)
Ensemble (Black Box) PGD ϵ = 4/255, iter = 30	18.91	17.15 (-1.76)	26.6 (+7.69)	30.35 (+11.44)	31.89 (+12.98)	40.19 (+21.28)
Square Attack (Black Box) ϵ = 4/255, queries = 1000	9.29	36.47 (+27.18)	73.79 (+64.50)	71.18 (+61.89)	75.85 (+66.56)	68.84 (+59.55)
White Box PGD ϵ=1/255, iter = 30	19.64	17.56 (-2.08)	46.12 (+26.48)	54.98 (+35.34)	55.29 (+35.65)	64.26 (+44.62)
White Box PGD ϵ=2/255, iter = 30	0.51	0.45 (-0.06)	8.51 (+8.00)	17.22 (+16.71)	14.94 (+14.34)	44.85 (+44.34)
		CIFAR-100 (ResNet-32) (test samples = 10000)				
Clean	71.42	63.89 (-7.53)	62.44 (-8.98)	55.48 (-15.94)	64.20 (-7.22)	44.41 (-27.01)
Transfer (ResNet-20) PGD ϵ=6/255, iter = 30	9.61	8.45 (-1.16)	11.14 (+1.53)	11.83 (+2.22)	14.88 (+5.27)	15.76 (+6.15)
Ensemble (Black Box) PGD ϵ=4/255, iter = 30	9.88	8.03 (-1.85)	11.95 (+2.07)	12.59 (+2.71)	17.07 (+7.19)	17.60 (+7.72)
Square Attack (Black Box) ϵ = 4/255, queries = 1000	2.76	32.33 (+29.57)	43.59(+40.83)	38.12 (+35.36)	48.28 (+45.52)	35.25 (+32.49)
White Box PGD ϵ=1/255, iter 30	5.78	6.53 (+0.75)	24.22 (+18.44)	28.47 (+22.69)	30.45 (+24.67)	32.4 (+26.62)
White Box PGD ϵ=2/255, iter 30	0.24	0.39 (+0.15)	4.55 (+4.31)	8.27 (+8.03)	8.94 (+8.70)	20.14 (+19.9)
		ImageNet (ResNet-18) (test samples = 1000)				
Clean	69.56	65.2 (-4.36)	64.9 (-4.66)	62.5 (-7.06)	67.1 (-2.46)	65.1 (-4.46)
Square Attack (Black Box) ϵ = 4/255, queries = 500	35.70	49.20 (+13.50)	55.40 (+19.70)	53.10 (+17.40)	56.90 (21.20)	46.10 (+10.40)
White Box PGD ϵ=1/255 , iter = 30	0.40	0.60 (+0.20)	4.50 (+4.10)	10.30 (+9.90)	9.6 (+9.20)	44.3 (+43.90)
White Box PGD ϵ=2/255, iter = 30	0.10	0.10 (+0.00)	0.20 (+0.10)	0.50 (+0.40)	0.10 (+0.00)	33.50 (+33.40)

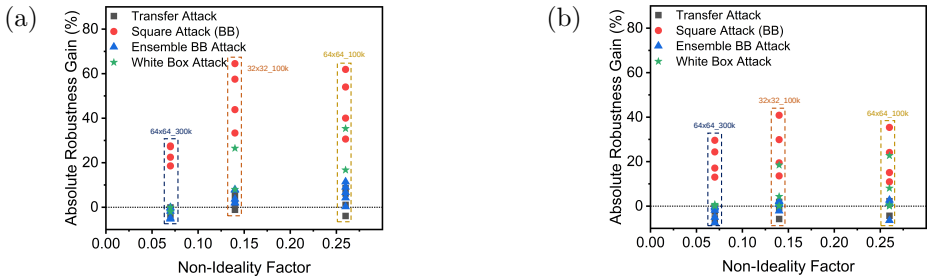


Figure 5.6. : Hardware-in-Loop Adaptive Black Box Attacks (PGD, iter=30) on CIFAR-10/100. Target NVM model is 64x64_100k, and the attacks are generated using 3 different NVM models.

gain. We observed an increase in the absolute improvement from baseline accuracy as we move from Transfer attacks to Ensemble Black Box to White Box attacks.

- The resulting accuracy is a combination of two opposing forces. The errors caused by the non-idealities try to lower the accuracy, while the intrinsic robustness arising from the same non-idealities lower the effectiveness of the attack and pushes the accuracy higher than the baseline. For example, for 64x64_300k (NF = 0.07), the MVM operations are close to ideal computation for the non-adaptive attacks to transfer successfully. Whereas, the more non-ideal crossbar models, 32x32_100k and 64x64_100k, have greater clean accuracy degradation due to functional errors, but have higher adversarial accuracy, as the non-idealities hinder the transfer of the attacks. This see-saw

effect can also be seen in Fig. 5.6, where we plot the robustness gain vs non-ideality factor (NF) of crossbars for all the non-adaptive attacks. We see a significant difference as NF increases from 64x64_300k to 32x32_100k. At 64x64_100k, we see the gain taper slightly below 32x32_100k, as inaccurate computations start to have a greater impact over intrinsic robustness.

- Overall, the intrinsic robustness of NVM crossbars is often within the ball park of Input BW Reduction. However, stronger adversarial defenses such as SAP [94] and Random Padding [103] have performed much better.

5.4.2 Hardware-in-Loop Adaptive Attacks

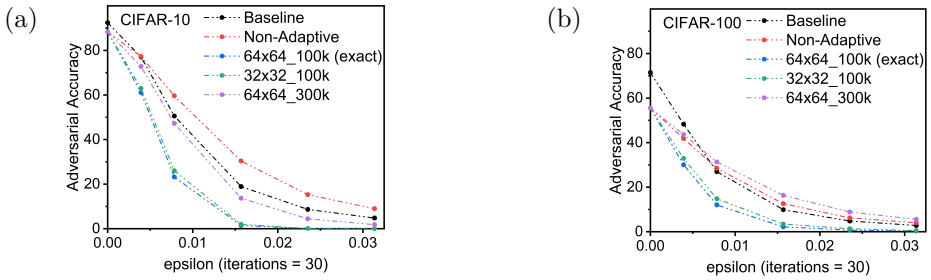


Figure 5.7. : Hardware-in-Loop Adaptive Black Box Attacks (PGD, iter=30) on CIFAR-10/100. Target NVM model is 64x64_100k, and the attacks are generated using 3 different NVM models.

Ensemble Black Box Attacks: When the attacker builds their synthetic dataset by querying the NVM crossbar hardware implementation of the DNN, the resulting Ensemble Black Box attacks are much more effective. The adversarial accuracy of the hardware falls significantly below the baseline, as shown in Fig. 5.7 and Table 5.4. Even when the attack is built using a crossbar model different from the target, accuracy degradation is significant. We observe that attacks generated using 32x32_100k (NF = 0.14) are stronger than those generated using 64x64_300k (NF = 0.07) when applied to 64x64_100k (NF= 0.26). This implies that the lesser the difference in NF, the more effective are the attacks.

Square Attack (Black Box): By repeatedly querying the actual NVM crossbar based hardware, the attacker could generate much stronger attacks, as shown in Fig. 5.4 and Table 5.4. In fact, the generated attacks are as strong as the baseline, however, when there is a

Table 5.4. : Hardware-in-Loop Adaptive Attacks

				NVM Crossbar Model (Target)		
Dataset	Test Samples	Attack ϵ	Baseline	64×64_300k	32×32_100k	64×64_100k
Ensemble BB Attack (iter=30)				Attacker's NVM Crossbar model:		64×64_100k
CIFAR-10	10000	4/255	18.91	1.95 (-16.96)	1.45 (-17.46)	**1.27** (-17.64)
CIFAR-100	10000	4/255	9.88	8.54 (-1.34)	2.74 (-7.74)	**2.17** (-7.71)
Square Attack (BB) (queries=30)				Attacker's NVM Crossbar model:		32×32_100k
CIFAR-10	1000	8/255	67.50	71.80 (+4.30)	**66.60** (-0.90)	64.10 (-3.40)
CIFAR-100	1000	8/255	40.10	49.20 (+9.1)	**32.50** (-7.60)	26.70 (-13.40)
Imagenet	1000	8/255	48.50	53.30 (+4.80)	**46.00** (-2.50)	44.30 (-4.20)
White Box PGD (iter=30)				Attacker's NVM Crossbar model:		64×64_100k
CIFAR-10	10000	1/255	19.64	43.45 (+23.81)	31.78 (+12.14)	**28.84** (+9.2)
CIFAR-10	10000	2/255	0.51	6.98 (+6.47)	2.13 (+1.62)	**1.87** (+1.36)
CIFAR-100	10000	1/255	5.78	28.21 (+22.43)	10.86 (+5.08)	**9.73** (+3.95)
ImageNet	1000	1/255	0.40	–	–	**0.80** (+0.40)

significant mismatch in hardware properties, the attack doesn't transfer well, as in the case of 64x64_300k as the defending crossbar model.

White Box Attacks: The results for hardware in loop White Box attacks are presented in Table 5.4. The values in bold indicate that attacker's NVM crossbar model is an exact match to the target model's underlying hardware. Even when the attacker has full knowledge of the hardware, the non-idealities help improve robustness. We observe that if the attacker's NVM model is different from the target, the attacks do not transfer well and are weaker than non-adaptive attacks. For example, for CIFAR-10, under attack epsilon $\epsilon = 1/255$, the accuracy of 64x64_300k NVM model is 0.60% for a non-adaptive attack, but 43.45% for an adaptive attack with incorrect NVM model. Thus having an incorrect crossbar model is worse than having no model at all in this case.

5.5 Conclusion

Non-idealities in NVM crossbars have been a long-standing challenge [90] affecting the feasibility of analog computing hardware, and several techniques have been proposed to compensate for it [92]. In this work, we study these non-idealities from the new perspective of adversarial robustness. We observed that DNNs implemented on an NVM crossbar

hardware exhibit increased adversarial robustness under varied threat models. While this robustness falls short of other defenses [94], [103], [111], an important point to note is that such robustness is intrinsic to the NVM crossbar hardware, unlike other defenses which have a computational overhead. Also, any algorithmic defense can be further implemented on the analog hardware for additional robustness. The non-ideality factor (NF) of the crossbar model determines the degree of robustness, therefore, one can potentially design NVM crossbars with optimal trade-off between accuracy degradation and increased robustness due to non-idealilties. We have demonstrated "Hardware-in-Loop" attacks where the knowledge of underlying hardware helps generate stronger attacks. While we have considered NVM crossbar models based on RRAM technology [9], analog hardware based on other technologies [10], [88] are also possible. This, along with chip to chip variations, may further hinder the transferability of attacks generated on one analog computing hardware to another. In summary, this work is the first step toward understanding the role of non-idealities in NVM crossbar hardware for adversarial robustness. It opens the possibilities of defenses that leverage the non-ideal computations, and on the other hand, attacks that exploit these non-idealities.

6. NOISE STABILITY AND ROBUSTNESS OF ADVERSARIALLY TRAINED NETWORKS ON NVM CROSSBARS

6.1 Introduction

In the previous chapter, we focused on quantifying the intrinsic robustness of NVM crossbars for vanilla networks, that is DNNs trained on unperturbed inputs. However, adversarial training of DNNs, i.e. training with adversarially perturbed images, is the benchmark technique for robustness, and sole reliance on intrinsic robustness of the hardware is not sufficient. In this chapter, we explore the design of robust DNNs through the amalgamation of adversarial training and the intrinsic robustness offered by NVM crossbar based analog hardware.

When an MVM operation is executed in an NVM crossbar, the output is sensed as a summation of currents through resistive NVM devices. The non-ideal behaviour of the crossbar and its peripheral circuits results in error in the final calculation, and overall performance degradation of the DNN [90], [92]. As these deviations depend on multiple analog variables (voltages, currents, resistances), they are nearly impossible to estimate without a complete model of the architecture and the device parameters, which are difficult for an attacker to obtain. In Ch. 6, we demonstrated that this non-ideal behaviour can be utilized to obscure the true gradients of the DNN and provide robustness against various types of adversarial attacks. Their analyses of the benefit of intrinsic robustness was limited to vanilla DNNs, i.e. networks trained on the unperturbed, or "clean", images. However, for a robust DNN implementation, adversarial training is essential. Hence, in this chapter, we study the performance and robustness of adversarially trained networks when implemented on such NVM crossbars. We observe that adversarial training reduces the noise stability of DNNs, i.e. its ability to withstand perturbations within the network during inference. We also note that the robustness gain from the crossbar is applicable only for certain degrees of attack perturbations. To summarize, our main contributions are:

- We explored the challenges of designing adversarially robust DNNs through the amalgamation of adversarial training and the intrinsic robustness offered by NVM crossbar based analog computing hardware.
- We analyzed the performance of adversarially trained DNNs on NVM crossbars for unperturbed images. Compared against vanilla DNNs (networks trained on unperturbed data), we observed that adversarially trained networks are less noise stable, and hence, suffer greater performance degradation on NVM crossbar based analog hardware.
- For Non-Adaptive Projected-Gradient-Descent White-Box Attacks[112], we demonstrated that the non-idealities provide a gain in robustness when the epsilon of the adversarial attack (ϵ_{attack}, the degree of input perturbations) is greater than the epsilon of the adversarial training (ϵ_{train}). In fact, with careful co-design, one can implement a DNN trained with lower ϵ_{train} that will have the same or even higher robustness than a DNN trained with a higher ϵ_{train} while maintaining a higher natural test accuracy.

The rest of the chapter is organized as follows. In Sec. 6.2, we provide an overview of the prior works on robustness using analog computing. In Sec. 6.3, we describe in detail the evaluation framework used in our experiments. This is followed by Sec. 6.4, where we report our findings and discuss the implications. Finally, in Sec. 6.5, we provide the conclusions of our findings.

6.2 Related Work

Prior works have shown that various facets of analog computing can be incorporated for adversarial robustness. In [113], the authors use an Optical Processor Unit (OPU) as an analog defense layer. It performs a fixed random transformation, and obfuscates the gradients and the parameters, to achieve adversarial robustness. While [113] used analog processing as an extra layer in their computation, [114] implemented neural networks on an NVM crossbar based analog hardware, and analyzed the impact on accuracy and robustness. They demonstrated two types of Adversarial attacks, one where the attacker is unaware of the hardware (Non-Adaptive Attacks), and the second being Hardware-in-Loop attacks, where the attacker has access to the NVM crossbar. They showed that during Non-Adaptive

Attacks on a vanilla DNN, i.e. a network trained on unperturbed images with no other defenses, the gradient obfuscation by the analog hardware provides substantial robustness against both Black Box and White Box Attacks. However, once the attacker gains complete access to the hardware, it can generate Hardware-in-Loop attacks and this significantly diminishes the robustness gain observed earlier. In this chapter, we further expand the analysis of Non-Adaptive Attacks on adversarially trained DNNs implemented on NVM crossbars.

6.3 Evaluating Adversarially Trained Networks on NVM Crossbars

Table 6.1. : ResNet Architectures used for CIFAR-10/100

		CIFAR-10		CIFAR-100	
Group Name	Output Size	ResNet10w1	ResNet10w4	ResNet20w1	ResNet20w4
conv0	32×32	$3 \times 3, 16$	$3 \times 3, 16$	$3 \times 3, 16$	$3 \times 3, 16$
conv1	32×32	$\begin{bmatrix} 3 \times 3, 16 \\ 3 \times 3, 16 \end{bmatrix} \times 1$	$\begin{bmatrix} 3 \times 3, 64 \\ 3 \times 3, 64 \end{bmatrix} \times 1$	$\begin{bmatrix} 3 \times 3, 16 \\ 3 \times 3, 16 \end{bmatrix} \times 3$	$\begin{bmatrix} 3 \times 3, 64 \\ 3 \times 3, 64 \end{bmatrix} \times 3$
conv2	16×16	$\begin{bmatrix} 3 \times 3, 32 \\ 3 \times 3, 32 \end{bmatrix} \times 1$	$\begin{bmatrix} 3 \times 3, 128 \\ 3 \times 3, 128 \end{bmatrix} \times 1$	$\begin{bmatrix} 3 \times 3, 32 \\ 3 \times 3, 32 \end{bmatrix} \times 3$	$\begin{bmatrix} 3 \times 3, 128 \\ 3 \times 3, 128 \end{bmatrix} \times 3$
conv3	8×8	$\begin{bmatrix} 3 \times 3, 64 \\ 3 \times 3, 64 \end{bmatrix} \times 2$	$\begin{bmatrix} 3 \times 3, 256 \\ 3 \times 3, 256 \end{bmatrix} \times 2$	$\begin{bmatrix} 3 \times 3, 64 \\ 3 \times 3, 64 \end{bmatrix} \times 3$	$\begin{bmatrix} 3 \times 3, 256 \\ 3 \times 3, 256 \end{bmatrix} \times 3$
Linear	1×1	Avg Pool 64×10	Avg Pool 256×10	Avg Pool 64×100	Avg Pool 256×100

In this work, we implement adversarially trained DNNs on NVM crossbars based analog hardware and analyze their performance. We first study the effect on natural accuracy, i.e. their performance in the absence of any attack. Analog computations are non-ideal in nature, and the degree of performance degradation depends on the resilience of the DNN to internal perturbations at inference. The degree of performance degradation helps us estimate the noise stability of adversarially trained networks, and understand how they differ from regular networks. Next, we evaluate the performance of these DNNs under adversarial attack and identify the benefit offered by NVM crossbars. Our experimental setup is described in detail in this section.

6.3.1 Datasets and Network Models

We perform our training and evaluation on two image recognition tasks:

- **CIFAR-10** [60]: It is a dataset containing $50,000$ training images, and $10,000$ test images, each of dimension 32×32 across 3 RGB channels. There are total 10 classes. We use two different network architectures for CIFAR-10, a 10-layer ResNet [67], ResNet10w1 and a $4\times$ inflated version of it, Resnet10w4 (Table 6.1).
- **CIFAR-100** [60]: This dataset also contains $50,000$ training images, and $10,000$ test images, each of dimension 32×32 across 3 RGB channels, and there are total 100 classes. We use two different network architectures for CIFAR-100, a 20-layer ResNet [67], ResNet20w1 and a $4\times$ inflated version of it, ResNet20w4 (Table 6.1).

6.3.2 Adversarial Attacks

We evaluate the DNNs against Non-Adaptive Attacks, where the attacker has no knowledge of the analog hardware, and assumes the DNN is implemented on accurate digital hardware. There exists a variety of attacks depending on how much information about the DNN is available to the attacker. The strongest category of attacks is White-Box Attacks, where the attacker has full knowledge of the DNN weights, inputs and outputs, and we chose this scenario to test our adversarially trained DNNs. We use Projected Gradient Descent (PGD) [112] to generate l_∞ norm bound iterative perturbations, per the eq.:

$$x^{t+1} = \Pi_{x+S}(x^t + \alpha sgn(\nabla_x L(\theta, x^t, y)) \tag{6.1}$$

The adversarial example generated at $(t+1)^{th}$ iteration is represented as x^{t+1}. $L(\theta, x, y)$ is the DNN's cost function. It depends on the DNN parameters θ, input x, and labels y. S is the set of allowed perturbations, and is defined using the the attack epsilon (ϵ_{attack}) as $S = \Big(\delta | \big(x+\delta \geq max(x+\epsilon_{attack}, 0)\big) \wedge \big(x+\delta \leq min(x+\epsilon_{attack}, 255)\big)\Big)$, where $x \in [0, 255]$. The input is an 8-bit image, and every pixel is a value between 0 to 255. For all our experiments, we set the number of iterations to 50. The ϵ_{attack} is set within the range $[2, 16]$ and signifies the maximum distortion that can be added to any pixel.

6.3.3 Adversarial Training

First, we train our DNNs on the unperturbed dataset, i.e. "clean" images. We train two DNNs, ResNet10w1 and ResNet10w4 on CIFAR-10, and two DNNs, ResNet20w1 and ResNet20w4 on CIFAR-100. These 4 DNNs are referred to as vanilla DNNs, and form the baseline for our later experiments. Next, we generate adversarially trained DNNs, by using iterative PGD training [112]. At every training step, the batch of clean images is iterated over the network several times to generate a batch of adversarial images, which is then used to update the weights of the network. For a single training procedure, the epsilon of the adversarial image (ϵ_{train}) is fixed, and the iterations is set to 50. For each network architecture in Table 6.1, we generate 4 DNNs, each trained with a different ϵ_{train} (=[2,4,6,8]) for 200 epochs.

6.3.4 Emulation of the Analog Hardware

Table 6.2. : NVM Crossbar Model Description [90]

	Crossbar parameters		
NVM Crossbar Model	Size	R_{ON} (Ω)	NF
32×32_100k	32×32	100k	0.14
64×64_100k	64×64	100k	0.26

Table 6.3. : Functional Simulator precision parameters

Simulation Parameters	CIFAR-10	CIFAR-100
I_w	4	4
W_w	4	4
I_{bit}	16	16
W_{bit}	16	16
I_{i-bit}	13	12
W_{i-bit}	13	12
O_{bit}	32	32

In order to evaluate the performance of our adversarially trained DNNs on NVM crossbars in the presence of non-idealities, we need to a) model Equation 5.2 to express the transfer characteristics of NVM crossbars and b) map the convolutional and fully-connected layers of the workload on a typical spatial NVM crossbar architecture such as PUMA [89]. The modelling of Eq. 5.2 is performed by considering the GENIEx [90] crossbar modeling technique which uses a 2-layer perceptron network where the inputs to the network are concatenated V and G vectors and the outputs are the non-ideal current, I_{ni}. The perceptron network is trained using data obtained from HSPICE simulations of NVM crossbars considering the aforementioned non-idealities with different combinations of V and G vectors and matrices, respectively.

We generate two crossbar models, 32x32_100k and 64x64_100k, their parameters given in Table 6.2. The NVM device used in the crossbar was the the RRAM device model [115]. Each crossbar model has a Non-ideality Factor, defined as,

$$NF = Average\left(\frac{Output_{ideal} - Output_{non-ideal}}{Output_{ideal}}\right) \tag{6.2}$$

In our experiments, we consider two crossbar models of different sizes. Higher crossbar size results in higher deviations in computations.

Next, we use a simulation framework, proposed in [90], to map DNN layers on NVM crossbars. Afterwards, we integrate these NVM crossbar models with our PyTorch framework, using the PUMA functional simulator [89], [90], and map DNN layers on NVM crossbars. Such a mapping consists of three segments: i) Lowering, ii) Tiling and iii) Bit Slicing. Lowering refers to dividing the convolutional or linear layer operation into individual MVM operation; Tiling involves distributing bigger MVM operations into smaller crossbar-sized MVM sub-operations. Finally, since each NVM device can only hold upto a limited number of discrete levels, bit-slicing is performed to accommodate MVM operations using large bit-precision inputs and weights. This is done by dividing them into smaller chunks called streams (for inputs) and slices (for weights). We set all precision parameters, given in Table 6.3, for 16 bit fixed point operations, and ensure that for an ideal crossbar behaviour, there is neglible accuracy loss due to reduction in precision. Thus, in our experiments with NVM

crossbars any change in performance is solely due to non-ideal behaviour. The bit slicing and fixed-point precision parameters are defined as:

- Input Stream Width (I_w) - Bit width of input fragments after slicing.
- Weight Slice Width (W_w) - Bit width of weight fragments after slicing.
- Input precision (I_{bit}) - Fixed-point precision of the inputs. This is divided into integer bits (I_{i-bit}) and fractional bits (I_{f-bit}).
- Weight precision (W_{bit}) - This refers to fixed-point precision of the weights. This is divided into integer bits and fractional bits (W_{f-bit}).
- Output precision (O_{bit}) - Fixed-point precision of the outputs of MVM computations.

6.4 Results

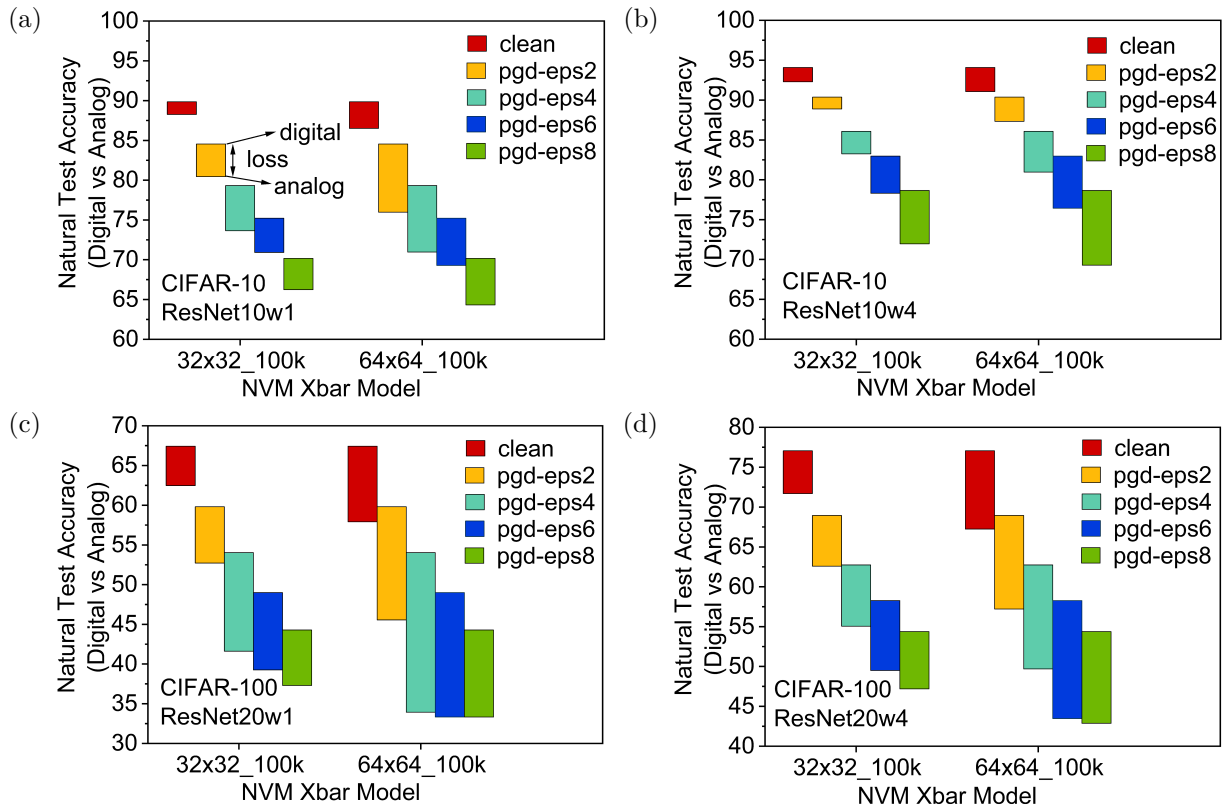


Figure 6.1. : Digital vs Analog Natural Test Accuracy for vanilla and adversarially trained DNNs. *clean*: vanilla training with unperturbed images. *pgd-epsN*: PGD adversarial training with $\epsilon_{train} = N = [2, 4, 6, 8]$ and iter $= 50$

6.4.1 Noise Stability of Adversarially Trained Networks

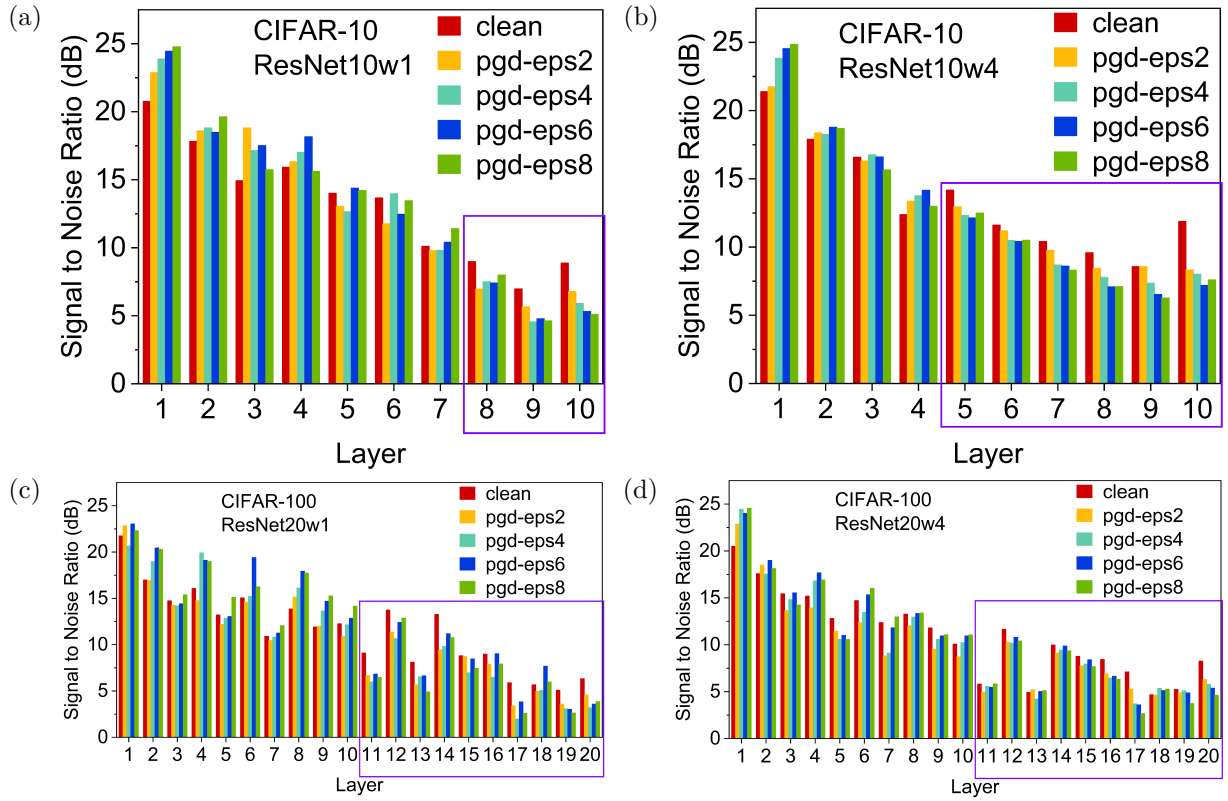


Figure 6.2. : Signal to Noise (SNR) at the output of every layer. for vanilla and adversarially trained DNNs. *clean*: vanilla training with unperturbed images. *pgd-epsN*: PGD adversarial training with $\epsilon_{train} = N = [2, 4, 6, 8]$ and iter $= 50$. NVM crossbar model: 64x64_100k ($NF = 0.26$).

At first, we analyze the performance of the vanilla and adversarially trained DNNs on NVM crossbars in the absence of any adversarial attack. We define this accuracy as the Natural Test Accuracy of the DNNs. In Fig. 6.1, we present a floating column chart where the top of a column represents the accuracy on accurate digital hardware, and where the bottom of the column represents the accuracy on the NVM hardware. The length of the column signifies the drop in accuracy due to non-idealities. Adversarial training, in itself, reduces the natural test accuracy of a DNN, and higher the training epsilon (ϵ_{train}), lower is the test accuracy. When implemented on 2 different NVM crossbar models, 32x32_100k and 64x64_100k, the adversarially trained DNNs for all 4 network architectures suffer far greater accuracy degradation than their vanilla counterparts (i.e. DNNs trained on "clean" images).

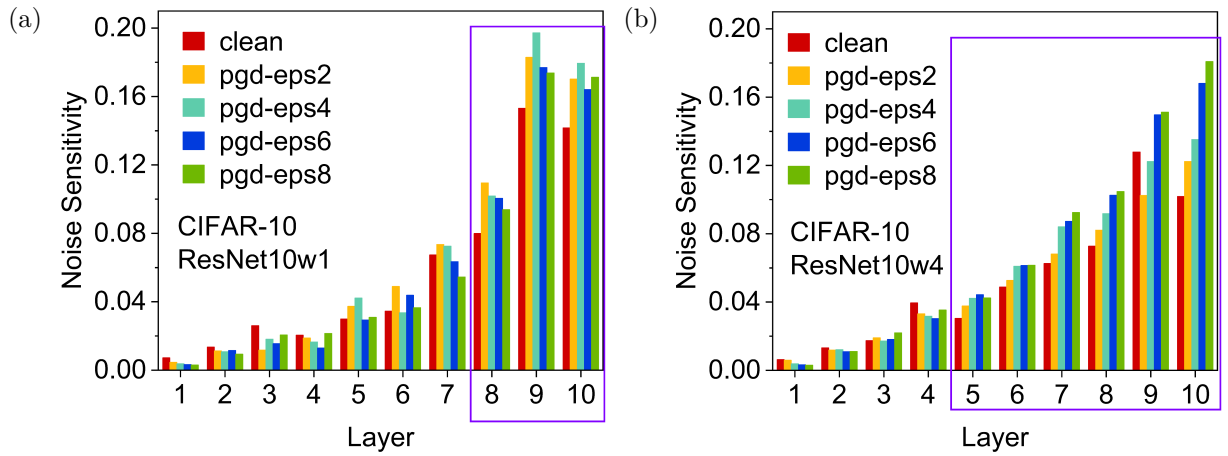


Figure 6.3. : Noise Sensitivity at the output of every layer. for vanilla and adversarially trained DNNs. *clean*: vanilla training with unperturbed images. *pgd-epsN*: PGD adversarial training with $\epsilon_{train} = N = [2, 4, 6, 8]$ and iter $= 50$. NVM crossbar model: 64x64_100k ($NF = 0.26$).

Also, according to Table 6.2, 32x32_100k has a lower non-ideality factor (NF) compared to 64x64_100k, i.e. the smaller crossbar has lesser deviations than the larger crossbar. This translates into smaller accuracy drop for 32x32_100k compared to 64x64_100k.

On the NVM crossbar model, 32x32_100k, the average accuracy drop for vanilla DNNs is 3.4% with the maximum of 5.38% for Resnet20w4 on CIFAR-100 dataset (Fig. 6.1d). Whereas for adversarially trained DNNs, the average is 6.2%, and the maximum drop is 12.44% for ResNet20w1($\epsilon_{train} = 4$) on CIFAR-100 dataset (Fig. 6.1c).

Similarly, for NVM crossbar model, 64x64_100k, the average accuracy drop for vanilla DNNs is 6.4% with the maximum of 9.83% for Resnet20w4 on CIFAR-100 dataset (Fig. 6.1d). Whereas for adversarially trained DNNs, the average is 10.3%, and the maximum drop is 20.12% for ResNet20w1($\epsilon_{train} = 4$) on CIFAR-100 dataset (Fig. 6.1c).

Thus, on an average, adversarially trained DNNs suffer 2× as much performance degradation on NVM crossbar, when compared to vanilla DNNs. To further investigate the cause of this accuracy drop, we look at the output post convolution at every layer of the DNN. To

quantify the deviation from ideal, we define two new metrics. The first is Signal to Noise Ratio (SNR), and it is defined as,

$$SNR = log_{10}\Big(\sum_{N} \frac{|Z_{analog}|^2}{|Z_{digital} - Z_{analog}|^2}\Big) \tag{6.3}$$

Here, Z is the output of a layer before the activation function, i.e. the output right after the MVM operation. The numerator signifies the total signal strength, while the denominator accounts for the noise in the signal, which is the difference between the digital and NVM crossbar implementation.

The second metric is called Noise Sensitivity, NS, and uses the same definition as in [116],

$$NS = \sum_{N} \frac{|Z_{digital} - Z_{analog}|^2}{|Z_{digital}|^2} \tag{6.4}$$

To computer SNR and NS we randomly sampled 1000 (N) images, i.e. 10% of the total test set for both CIFAR-10 and CIFAR-100, and evaluated the DNNs on 64x64_100k NVM crossbar model as it has the higher non-ideality factor. In Fig. 6.2, we observe that for later layers, SNR of adversarially trained DNNs is lower than that of vanilla networks. In case of CIFAR-10, layers 8-10 of ResNet10w1, and layers 5-10 of ResNet10w4 had lower SNR for the adversarially trained versions compared to the vanilla version. Similarly, in case of CIFAR-100, layers 11-20 (except layer 18) of of both ResNet20w1 and ResNet20w4 have lower SNR for the adversarially trained versions. Within a a DNN, the later layers have greater impact on the classification scores, as they are the closest to the final linear classifier. The lower SNR indicates that the noise introduced by the NVM non-idealities distorts the MVM outputs for adversarially trained DNNs by a larger margin as compared to the vanilla DNNs. Similarly, in Fig. 6.3, we observe that the noise sensitivity, NS, of the corresponding later layers is higher for adversarially trained networks, correlating with the trends of SNR and accuracy.

Our observations are in line with theoretical and empirical studies of adversarially trained DNNs. The process of adversarial training creates weight transformations that are less noise stable, i.e. given some perturbations at the layer input, it generates greater deviations in

the layer output. The optimization landscape of adversarial loss has increased curvature and scattered gradients [117], [118]. The solutions, i.e DNN weights, achieved by the adversarial training often doesn't lie within a stable minima. If one assumes the non-ideal deviations in the NVM crossbar as changes in the weights, then for vanilla DNNs, with stable, flat minimas, these changes have small effect on the loss, and the accuracy drop is low. However, for an adversarially trained DNN, the same degree of changes results in a greater change in loss, and hence greater performance degradation. Thus, this work provides a hardware-backed validation of the complex loss landscape of adversarial training.

6.4.2 Adversarial Robustness of Analog NVM crossbars

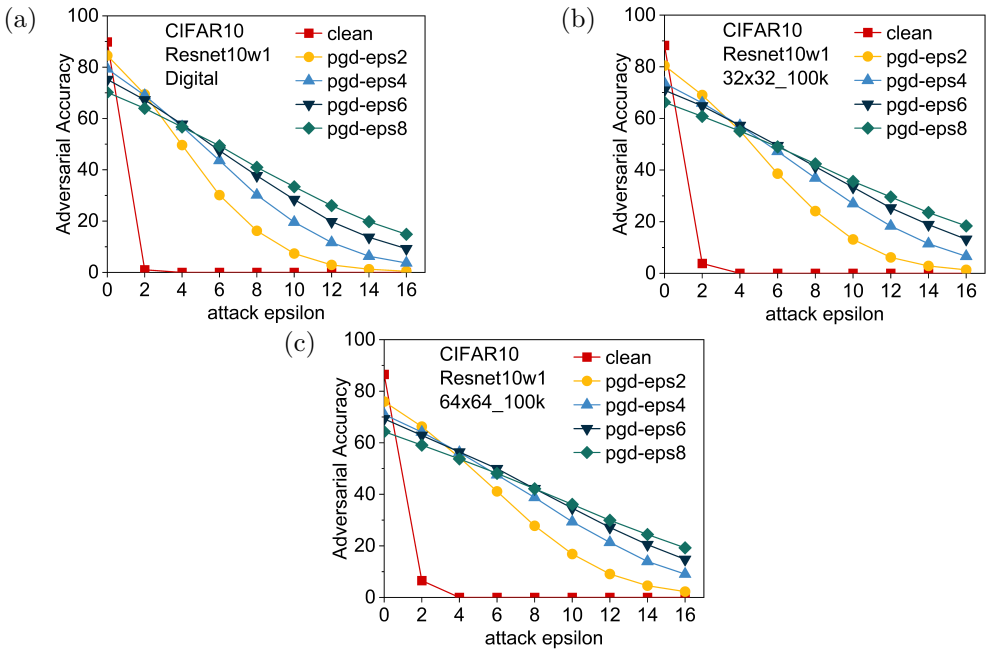


Figure 6.4. : Adversarial Accuracy under PGD White Box Attack (iter = 50) for ResNet10w1 architecture and CIFAR-10 implemented on 3 different hardware (a) Accurate Digital, (b) NVM crossbar model: 32x32_100k ($NF = 0.14$), and (c) NVM crossbar model: 64x64_100k ($NF = 0.26$). *clean*: vanilla training with unperturbed images. *pgd-epsN*: PGD adversarial training with $\epsilon_{train} = N = [2, 4, 6, 8]$ and iter = 50.

Next, we evaluate the DNNs under Non-Adaptive White Box Attack, where the attacker has complete information of the network weights, however they have no knowledge of the NVM crossbar. The attacker assumes the underlying hardware is accurate, and generates

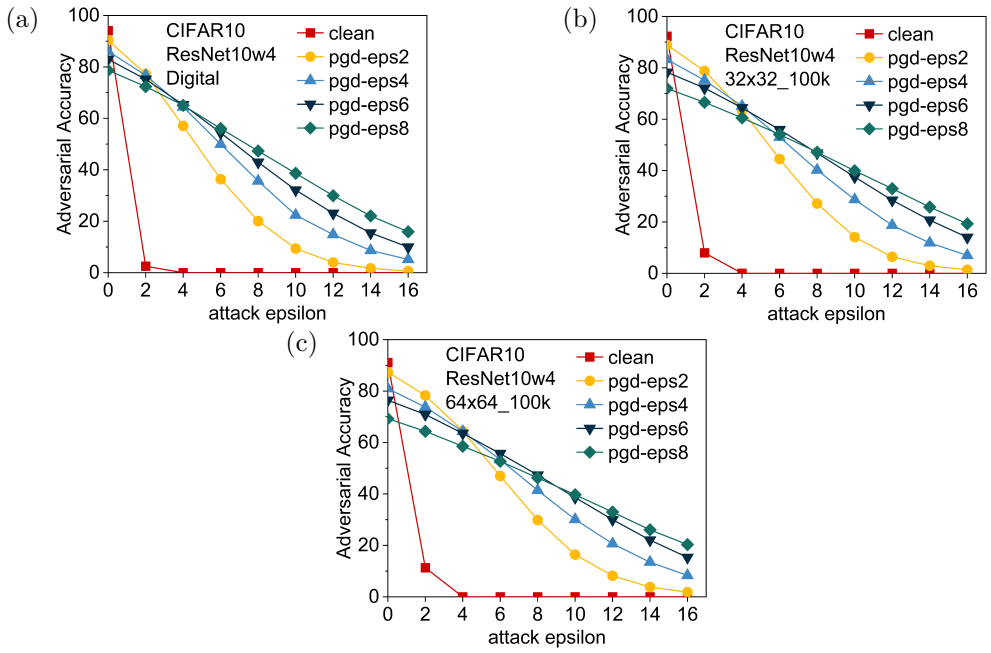


Figure 6.5. : Adversarial Accuracy under PGD White Box Attack (iter = 50) for ResNet10w4 architecture and CIFAR-10 implemented on 3 different hardware (a) Accurate Digital, (b) NVM crossbar model: 32x32_100k ($NF = 0.14$), and (c) NVM crossbar model: 64x64_100k ($NF = 0.26$). *clean*: vanilla training with unperturbed images. *pgd-epsN*: PGD adversarial training with $\epsilon_{train} = N = [2, 4, 6, 8]$ and iter = 50.

adversarial images using iterative PGD (iter =50). From Fig. 6.4-6.6, we see that adversarially trained DNNs perform better than vanilla DNNs for all values of ϵ_{attack}, and in every hardware. Thus, it is imperative to use adversarially trained DNNs to obtain robust implementations.

In Fig. 6.8 and 6.9, we plot the net gain in robustness, that is the difference in adversarial accuracy when the same DNN is implemented on an accurate digital hardware and NVM crossbar under the same adversarial attack condition. We observe that for a network trained with $\epsilon_{train} = N$, the intrinsic robustness of the network starts having a positive effect only in attacks where $\epsilon_{attack} > N$. At $\epsilon_{attack} = 0$, i.e in the absence of any attack, the DNNs' accuracy on the analog hardware is significantly lower than digital implementation. As ϵ_{attack} is increased, the difference in accuracy slowly reduces, and reaches a tipping point when the ϵ_{attack} is equal to ϵ_{train}. Beyond that, the analog implementation has a higher accuracy than

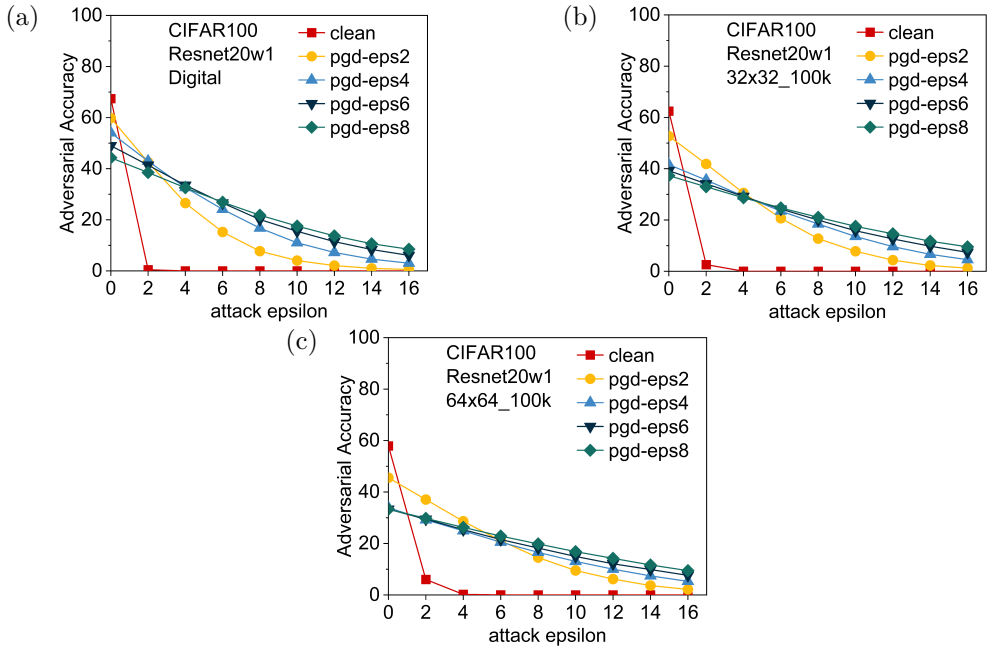


Figure 6.6. : Adversarial Accuracy under PGD White Box Attack (iter = 50) for ResNet20w1 architecture and CIFAR-100 implemented on 3 different hardware (a) Accurate Digital, (b) NVM crossbar model: 32x32_100k ($NF = 0.14$), and (c) NVM crossbar model: 64x64_100k ($NF = 0.26$). *clean*: vanilla training with unperturbed images. *pgd-epsN*: PGD adversarial training with $\epsilon_{train} = N = [2, 4, 6, 8]$ and iter = 50.

the digital, i.e. we are in the zone of robustness gain due to the non-idealities. We can draw the following conclusion:

- **The Push-Pull Effect**: As noted in [114], when under adversarial attack, the non-idealities have a dual effect on the accuracy. The first is the loss in accuracy because of inaccurate computations. And the second is the increase in accuracy due to gradient obfuscation. Because the hardware is unknown to the attacker, the adversarial attacks do not transfer fully to our implementation. At lower ϵ_{attack}, the loss due to inaccurate computations is higher, while at higher ϵ_{attack}, the intrinsic robustness becomes more significant.
- **Out of Distribution Samples**: In adversarial training, we train DNNs on images with adversarial perturbations bound by the training ϵ_{train}. During inference, adversarial images with $\epsilon_{attack} < \epsilon_{train}$ would have a distribution similar to the training images, and the DNN behaves similar to a vanilla DNN inferring unperturbed images.

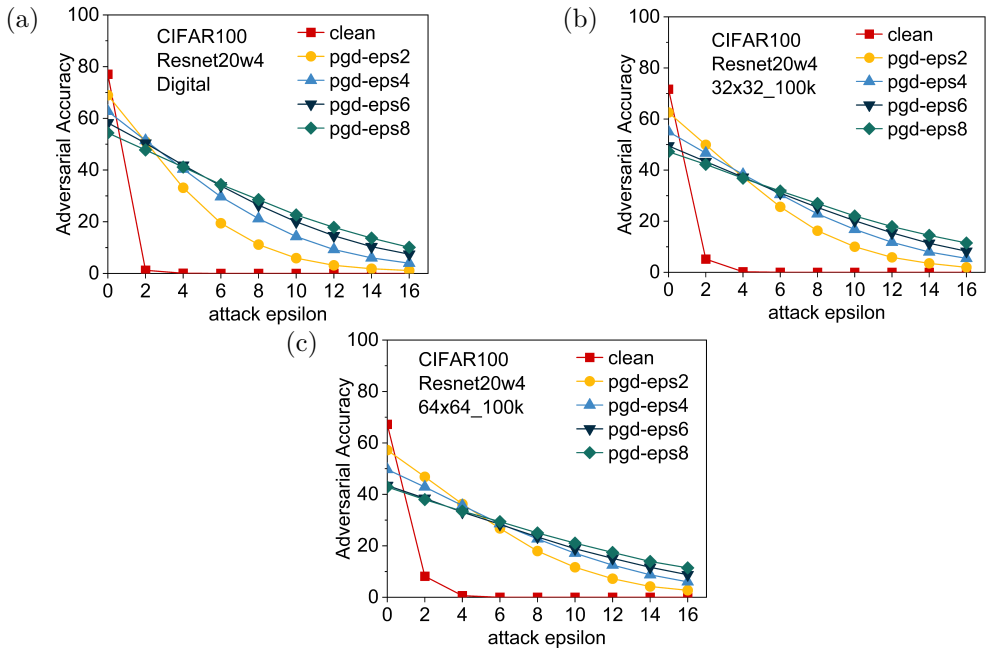


Figure 6.7. : Adversarial Accuracy under PGD White Box Attack (iter = 50) for ResNet20w4 architecture and CIFAR-100 implemented on 3 different hardware (a) Accurate Digital, (b) NVM crossbar model: 32x32_100k ($NF = 0.14$), and (c) NVM crossbar model: 64x64_100k ($NF = 0.26$). *clean*: vanilla training with unperturbed images. *pgd-epsN*: PGD adversarial training with $\epsilon_{train} = N = [2, 4, 6, 8]$ and iter = 50.

We only observe accuracy loss due to non-ideal computations. However, adversarial images with $\epsilon_{attack} > \epsilon_{train}$ are out of distribution with respect to the training data. Now, the DNN is under adversarial attack, and the intrinsic robustness of the NVM crossbar boosts the accuracy.

Design for Robust DNNs on NVM Crossbars: When designing robust DNNs on digital hardware, the tradeoff between the adversarial and the natural accuracy of a DNN is fairly straightforward. A higher ϵ_{train} during training leads to lowering of natural accuracy, and increase in adversarial accuracy, as shown in Fig. 6.4(a), 6.5(a), 6.6(a), and 6.7(a). However, the interplay of non-idealities and adversarial loss creates opportunities for design of DNNs where a DNN trained on lower ϵ_{train} can achieve both higher natural accuracy and higher adversarial accuracy, as shown in Fig. 6.6(b) and (c), and 6.7(b) and (c). For $\epsilon_{attack} = [2, 4]$ on 32x32_100k and for $\epsilon_{attack} = [2, 4, 6]$ on 64x64_100k, the ResNet20w1 DNN trained with $\epsilon_{train} = 2$ performs better or at par than the DNNs trained with higher ϵ_{train}. Thus, the

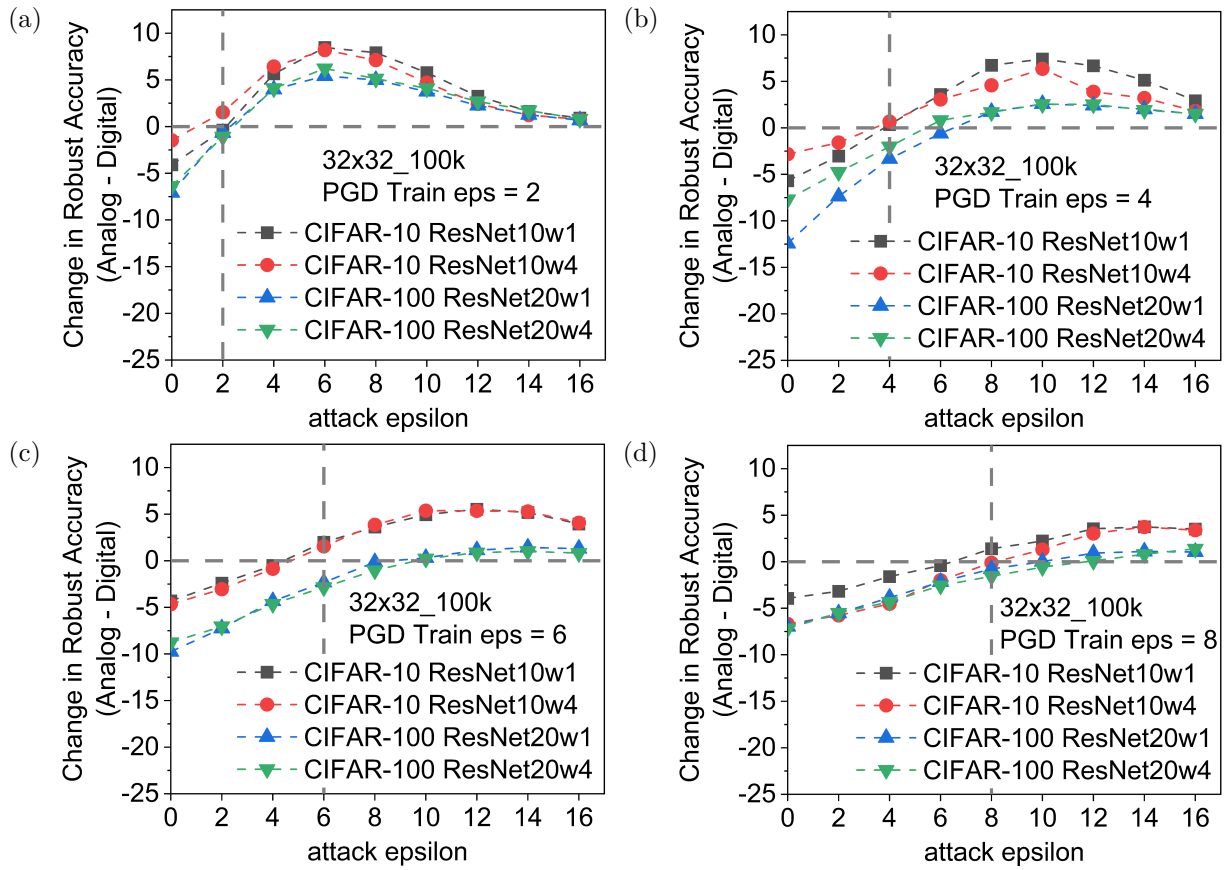


Figure 6.8. : Difference in Adversarial Accuracy (Robustness Gain = Analog - Digital) for varying PGD Attack (ϵ_{attack}). NVM crossbar model: 32x32_100k ($NF = 0.14$). 4 network architectures (ResNet10w1, ResNet10w4, ResNet20w1, ResNet20w4) on 2 datasets (CIFAR-10, CIFAR-100) are adversarially trained with (a) PGD, $\epsilon_{train} = 2$ and iter= 50 (b) PGD, $\epsilon_{train} = 4$ and iter= 50 (c) PGD, $\epsilon_{train} = 6$ and iter= 50 (d) PGD, $\epsilon_{train} = 8$ and iter= 50

non-idealities play a significant role in determining the best DNN within a desired robustness range.

6.5 Conclusion

NVM crossbars offer intrinsic robustness for defense against Adversarial attacks. While, prior works have quantified these benefits for vanilla DNNs, in this work, we propose design principles of robust DNNs for implementation on NVM crossbar based analog hardware by combining algorithmic defenses such as adversarial training and the intrinsic robustness of the analog hardware. First, we extensively analyze adversarially trained networks on

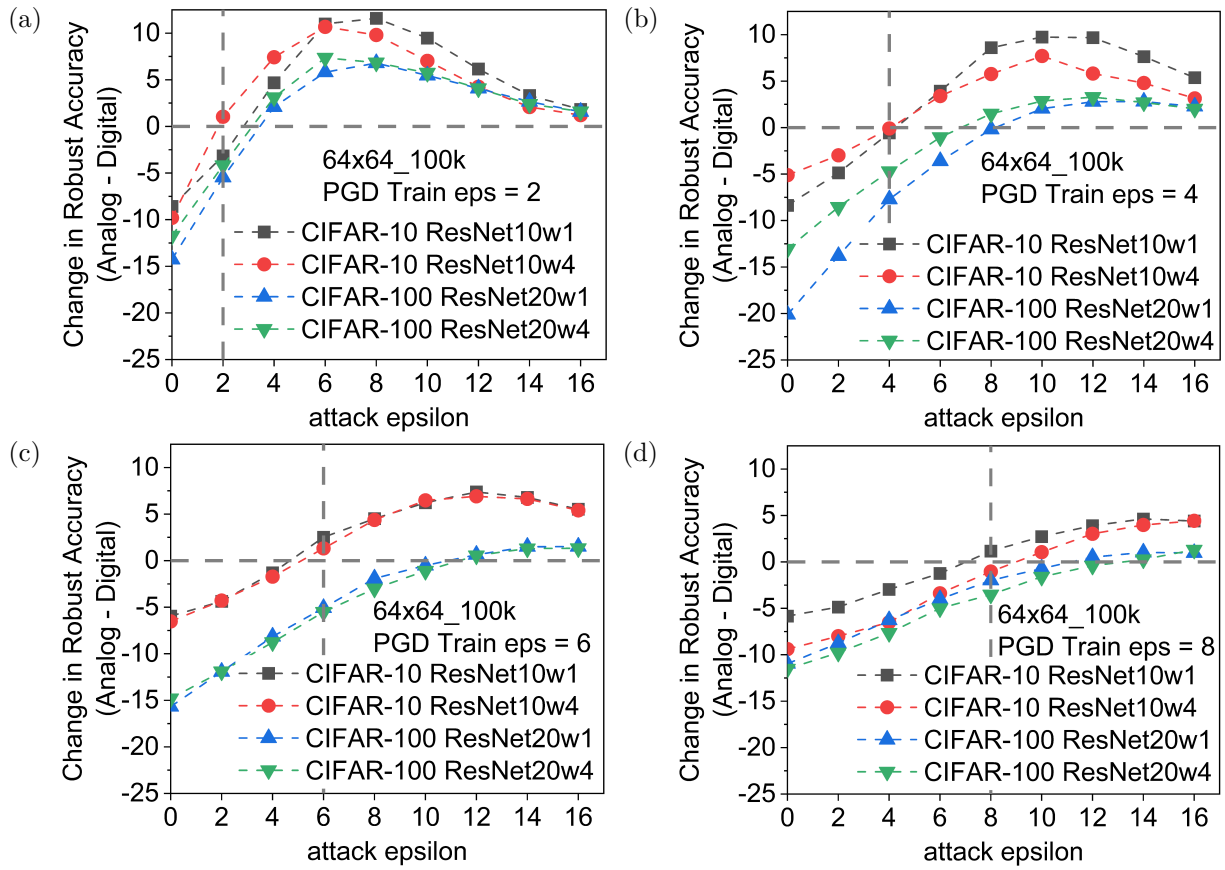


Figure 6.9. : Difference in Adversarial Accuracy (Robustness Gain = Digital - Analog) for varying PGD Attack (ϵ_{attack}). NVM crossbar model: 64x64_100k ($NF = 0.26$). 4 network architectures (ResNet10w1, ResNet10w4, ResNet20w1, ResNet20w4) on 2 datasets (CIFAR-10, CIFAR-100) are adversarially trained with (a) PGD, $\epsilon_{train} = 2$ and iter= 50 (b) PGD, $\epsilon_{train} = 4$ and iter= 50 (c) PGD, $\epsilon_{train} = 6$ and iter= 50 (d) PGD, $\epsilon_{train} = 8$ and iter= 50

NVM crossbars. We demonstrate that adversarially trained networks are less stable to the non-idealities of analog computing compared to vanilla networks, which impacts their classification accuracy on clean images. Next, we show that under adversarial attack, the gain in accuracy from the non-idealities is conditional on the epsilon of the attack (ϵ_{attack}) being greater than the epsilon used during adversarial training (ϵ_{train}). By implementing on NVM crossbar based analog hardware, a DNN trained with lower ϵ_{train} will have the same or even higher robustness than a DNN trained with a higher ϵ_{train} while maintaining a higher natural test accuracy. This work paves the way for a new paradigm of hardware-algorithm co-design which incorporates both energy-efficiency and adversarial robustness.

7. SUMMARY

In this thesis, we have analyzed the design and implementation of Neural Networks from both the algorithmic and hardware perspective. In the first half, we utilized the transferability of knowledge in Neural Networks and proposed new design techniques for multi-task and incremental learning. We designed the first audio-to-image synthesis model in Spiking Neural Networks (SNNs), and demonstrated that the spatio-temporal hidden representations of an SNN carry information than can be used across multiple modalities. We then proposed *Tree-CNN*, a hierarchical structure made of multiple CNNs that grows incrementally to learn new tasks. In the second half of the thesis, we explored algorithm-hardware co-design for efficient and robust implementations of Neural Networks. We trained deep binary stochastic neural networks and analyzed their performance under different binarization schemes, both for weights and activations. Such network design is very suitable for special purpose hardware based on emerging technologies. Next, we studied the impact on the robustness of the algorithms, when implemented on analog computing based hardware. We made the discovery that non-ideal computations, while detrimental to normal accuracy, improves adversarial performance by obfuscating the inference computations from the attacker. We analyzed both regular and adversarially trained DNNs, and observed that By implementing on NVM crossbar based analog hardware, a DNN trained with lower ϵ_{train} will have the same or even higher robustness than a DNN trained with a higher ϵ_{train} while maintaining a higher natural test accuracy. This work paves the way for a new paradigm of hardware-algorithm co-design which incorporates both energy-efficiency and adversarial robustness.

Future Work

Noise Stability of Adversarial Networks

During our experiments with adversarially trained networks on analog hardware, we observed that internal activations of adversarially trained networks have lower Signal-to-Noise Ratio (SNR), and are more sensitive to noise than vanilla networks. As a result, they suffer significantly higher performance degradation due to the non-ideal computations. This

begets further inquiry into the noise stability of adversarially trained networks, and what it means for the loss landscape of such networks. Further studies could provide valuable insight about adversarial training of deep neural networks.

Bio-Plausible Methods for Robustness and Efficiency

There have been recent advancements in crafting neural networks with activation distribution following closely to that of mammalian brain, such as COR-Net [119] and Vone-Net [120]. Such networks are promising in terms of both efficiency and adversarial robustness. These bio-plausible networks have certain unique properties, such as, they are shallower and wider, and use stochastic computation units and pre-defined convolutional filters. One could exploit these properties for improved design of neural network that are both energy efficient and robust.

REFERENCES

[1] K. Schwab, *The fourth industrial revolution*. Currency, 2017.

[2] 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.

[3] S. I. Serengil and A. Ozpinar, "Lightface: A hybrid deep face recognition framework," in *2020 Innovations in Intelligent Systems and Applications Conference (ASYU)*, IEEE, 2020.

[4] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *nature*, vol. 521, no. 7553, p. 436, 2015.

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

[6] I. J. Goodfellow, M. Mirza, D. Xiao, A. Courville, and Y. Bengio, "An empirical investigation of catastrophic forgetting in gradient-based neural networks," *arXiv preprint arXiv:1312.6211*, 2013.

[7] D. Roy, P. Panda, and K. Roy, "Synthesizing images from spatio-temporal representations using spike-based backpropagation," *Frontiers in Neuroscience*, vol. 13, p. 621, 2019.

[8] A. Shafiee *et al.*, "Isaac: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3, pp. 14–26, 2016.

[9] H.-S. P. Wong, H.-Y. Lee, S. Yu, Y.-S. Chen, Y. Wu, P.-S. Chen, B. Lee, F. T. Chen, and M.-J. Tsai, "Metal–oxide rram," *Proceedings of the IEEE*, vol. 100, no. 6, pp. 1951–1970, 2012.

[10] H.-S. P. Wong, S. Raoux, S. Kim, J. Liang, J. P. Reifenberg, B. Rajendran, M. Asheghi, and K. E. Goodson, "Phase change memory," *Proceedings of the IEEE*, vol. 98, no. 12, pp. 2201–2227, 2010.

[11] A. Sengupta, P. Panda, P. Wijesinghe, Y. Kim, and K. Roy, "Magnetic tunnel junction mimics stochastic cortical spiking neurons," *Scientific reports*, vol. 6, p. 30 039, 2016.

[12] A. Sengupta, G. Srinivasan, D. Roy, and K. Roy, "Stochastic inference and learning enabled by magnetic tunnel junctions," in *2018 IEEE International Electron Devices Meeting (IEDM)*, IEEE, 2018, pp. 15–6.

[13] A. Sengupta, M. Parsa, B. Han, and K. Roy, "Probabilistic deep spiking neural systems enabled by magnetic tunnel junction," *IEEE Transactions on Electron Devices*, vol. 63, no. 7, pp. 2963–2970, 2016.

[14] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus, "Intriguing properties of neural networks," *arXiv preprint arXiv:1312.6199*, 2013.

[15] P. Vincent, H. Larochelle, Y. Bengio, and P.-A. Manzagol, "Extracting and composing robust features with denoising autoencoders," in *Proceedings of the 25th international conference on Machine learning*, ACM, 2008, pp. 1096–1103.

[16] J. Masci, U. Meier, D. Cireşan, and J. Schmidhuber, "Stacked convolutional auto-encoders for hierarchical feature extraction," in *International Conference on Artificial Neural Networks*, Springer, 2011, pp. 52–59.

[17] 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.

[18] N. Srivastava and R. Salakhutdinov, "Learning representations for multimodal data with deep belief nets," in *International conference on machine learning workshop*, vol. 79, 2012.

[19] S. G. Wysoski, L. Benuskova, and N. Kasabov, "Evolving spiking neural networks for audiovisual information processing," *Neural Networks*, vol. 23, no. 7, pp. 819–835, 2010.

[20] N. Rathi and K. Roy, "Stdp-based unsupervised multimodal learning with cross-modal processing in spiking neural network," *IEEE Transactions on Emerging Topics in Computational Intelligence*, 2018.

[21] H. Markram, W. Gerstner, and P. J. Sjöström, "Spike-timing-dependent plasticity: A comprehensive overview," *Frontiers in synaptic neuroscience*, vol. 4, p. 2, 2012.

[22] K. S. Burbank, "Mirrored stdp implements autoencoder learning in a network of spiking neurons," *PLoS computational biology*, vol. 11, no. 12, e1004566, 2015.

[23] A. Tavanaei, T. Masquelier, and A. Maida, "Representation learning using event-based stdp," *Neural Networks*, 2018.

[24] 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.

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

[26] 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.

[27] P. J. Werbos, "Backpropagation through time: What it does and how to do it," *Proceedings of the IEEE*, vol. 78, no. 10, pp. 1550–1560, 1990.

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

[29] 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.

[30] 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.

[31] H. Xiao, K. Rasul, and R. Vollgraf, "Fashion-mnist: A novel image dataset for benchmarking machine learning algorithms," *arXiv preprint arXiv:1708.07747*, 2017.

[32] Q. Wu, M. McGinnity, L. Maguire, B. Glackin, and A. Belatreche, "Learning mechanisms in networks of spiking neurons," in *Trends in Neural Computation*, Springer, 2007, pp. 171–197.

[33] S. K. Esser, R. Appuswamy, P. Merolla, J. V. Arthur, and D. S. Modha, "Backpropagation for energy-efficient neuromorphic computing," in *Advances in Neural Information Processing Systems*, 2015, pp. 1117–1125.

[34] S. B. Shrestha and G. Orchard, "Slayer: Spike layer error reassignment in time," in *Advances in Neural Information Processing Systems*, 2018, pp. 1419–1428.

[35] B. Nessler, M. Pfeiffer, L. Buesing, and W. Maass, "Bayesian computation emerges in generic cortical microcircuits through spike-timing-dependent plasticity," *PLoS computational biology*, vol. 9, no. 4, e1003037, 2013.

[36] M. Benayoun, J. D. Cowan, W. van Drongelen, and E. Wallace, "Avalanches in a stochastic model of spiking neurons," *PLoS computational biology*, vol. 6, no. 7, e1000846, 2010.

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

[38] 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.

[39] M. Liberman, R. Amsler, K. Church, E. Fox, C. Hafner, J. Klavans, M. Marcus, B. Mercer, J. Pedersen, P. Roossin, *et al.*, "Ti 46-word," *Philadelphia (Pennsylvania): Linguistic Data Consortium*, 1993.

[40] M. Slaney, "Auditory toolbox," *Interval Research Corporation, Tech. Rep*, vol. 10, p. 1998, 1998.

[41] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, *et al.*, "Imagenet large scale visual recognition challenge," *International Journal of Computer Vision*, vol. 115, no. 3, pp. 211–252, 2015.

[42] 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," *MM '14 Proceedings of the ACM International Conference on Multime*, vol. d, pp. 177–186, 2014.

[43] D. Roy, P. Panda, and K. Roy, "Tree-cnn: A hierarchical deep convolutional neural network for incremental learning," *Neural Networks*, vol. 121, pp. 148–160, 2020.

[44] S. John Walker, *Big data: A revolution that will transform how we live, work, and think*, 2014.

[45] 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.

[46] R. Girshick, "Fast r-CNN," in *2015 IEEE International Conference on Computer Vision (ICCV)*, IEEE, 2015.

[47] K. Shmelkov, C. Schmid, and K. Alahari, "Incremental learning of object detectors without catastrophic forgetting," in *2017 IEEE International Conference on Computer Vision (ICCV)*, IEEE, 2017.

[48] Z. Li and D. Hoiem, "Learning without forgetting," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2017.

[49] R. Aljundi, P. Chakravarty, and T. Tuytelaars, "Expert gate: Lifelong learning with a network of experts," *CoRR, abs/1611.06194*, vol. 2, 2016.

[50] 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.

[51] S.-A. Rebuffi, A. Kolesnikov, and C. H. Lampert, "Icarl: Incremental classifier and representation learning," in *Proc. CVPR*, 2017.

[52] 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)*, IEEE, 2017, pp. 1–6.

[53] 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.

[54] 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.

[55] P. Panda and K. Roy, "Semantic driven hierarchical learning for energy-efficient image classification," in *2017 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, IEEE, 2017, pp. 1582–1587.

[56] 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.

[57] Z. Yan, H. Zhang, R. Piramuthu, V. Jagadeesh, D. DeCoste, W. Di, and Y. Yu, "HD-CNN: Hierarchical deep convolutional neural networks for large scale visual recognition," in *2015 IEEE International Conference on Computer Vision (ICCV)*, IEEE, 2015.

[58] N. Srivastava and R. R. Salakhutdinov, "Discriminative transfer learning with tree-based priors," in *Advances in Neural Information Processing Systems*, 2013, pp. 2094–2102.

[59] P. Kontschieder, M. Fiterau, A. Criminisi, and S. R. Bulo, "Deep neural decision forests," in *Computer Vision (ICCV), 2015 IEEE International Conference on*, IEEE, 2015, pp. 1467–1475.

[60] A. Krizhevsky, G. Hinton, *et al.*, "Learning multiple layers of features from tiny images," 2009.

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

[62] 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.

[63] MATLAB, *version 9.2.0 (R2017a)*. Natick, Massachusetts: The MathWorks Inc., 2017.

[64] I. J. Goodfellow, D. Warde-Farley, M. Mirza, A. Courville, and Y. Bengio, "Maxout networks," in *International Conference on Machine Learning*, 2013.

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

[66] L. Hertel, E. Barth, T. Kaster, and T. Martinetz, "Deep convolutional neural networks as generic feature extractors," in *2015 International Joint Conference on Neural Networks (IJCNN)*, IEEE, 2015.

[67] 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.

[68] Y. Cheng, D. Wang, P. Zhou, and T. Zhang, "A survey of model compression and acceleration for deep neural networks," *arXiv preprint arXiv:1710.09282*, 2017.

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

[70] 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.

[71] I. Garg, P. Panda, and K. Roy, "A low effort approach to structured cnn design using pca," *arXiv preprint arXiv:1812.06224*, 2018.

[72] S. Zhou, Y. Wu, Z. Ni, X. Zhou, H. Wen, and Y. Zou, "Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients," *arXiv preprint arXiv:1606.06160*, 2016.

[73] W. Severa, C. M. Vineyard, R. Dellana, S. J. Verzi, and J. B. Aimone, "Training deep neural networks for binary communication with the whetstone method," *Nature Machine Intelligence*, vol. 1, no. 2, p. 86, 2019.

[74] M. Courbariaux and Y. Bengio, "Binarynet: Training deep neural networks with weights and activations constrained to+ 1 or- 1. arxiv 2016," *arXiv preprint arXiv:1602.02830*,

[75] Rastegari *et al.*, "Xnor-net: Imagenet classification using binary convolutional neural networks," in *European Conference on Computer Vision*, Springer, 2016, pp. 525–542.

[76] T. Tuma, A. Pantazi, M. Le Gallo, A. Sebastian, and E. Eleftheriou, "Stochastic phase-change neurons," *Nature nanotechnology*, vol. 11, no. 8, p. 693, 2016.

[77] D. Roy, I. Chakraborty, and K. Roy, "Scaling deep spiking neural networks with binary stochastic activations," in *2019 IEEE International Conference on Cognitive Computing (ICCC)*, IEEE, 2019, pp. 50–58.

[78] M. Pfeiffer and T. Pfeil, "Deep learning with spiking neurons: Opportunities and challenges," *Frontiers in neuroscience*, vol. 12, 2018.

[79] G. E. Hinton, D. Rumelhart, and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, no. 9, pp. 533–536, 1986.

[80] R. Collobert, K. Kavukcuoglu, and C. Farabet, "Torch7: A matlab-like environment for machine learning," in *BigLearn, NIPS Workshop*, 2011.

[81] S. Hochreiter, Y. Bengio, P. Frasconi, J. Schmidhuber, *et al.*, *Gradient flow in recurrent nets: The difficulty of learning long-term dependencies*, 2001.

[82] Keckler *et al.*, "Gpus and the future of parallel computing," *IEEE Micro*, no. 5, pp. 7–17, 2011.

[83] 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.

[84] N. P. Jouppi *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *2017 ACM/IEEE 44th Annual ISCA*, IEEE, 2017, pp. 1–12.

[85] E. Chung, J. Fowers, K. Ovtcharov, M. Papamichael, A. Caulfield, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, *et al.*, "Serving dnns in real time at datacenter scale with project brainwave," *iEEE Micro*, vol. 38, no. 2, pp. 8–20, 2018.

[86] "Nvidia tesla v100 gpu architecture, the world's most advanced data center gpu," NVIDIA Corporation, Tech. Rep., 2017.

[87] X. Xu *et al.*, "Scaling for edge inference of deep neural networks," *Nature Electronics*, vol. 1, no. 4, pp. 216–222, 2018.

[88] X. Fong, Y. Kim, K. Yogendra, D. Fan, A. Sengupta, A. Raghunathan, and K. Roy, "Spin-transfer torque devices for logic and memory: Prospects and perspectives," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 1, pp. 1–22, 2015.

[89] A. Ankit, I. E. Hajj, S. R. Chalamalasetti, G. Ndu, M. Foltin, R. S. Williams, P. Faraboschi, W.-m. W. Hwu, J. P. Strachan, K. Roy, *et al.*, "Puma: A programmable ultra-efficient memristor-based accelerator for machine learning inference," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019, pp. 715–731.

[90] I. Chakraborty, M. F. Ali, D. E. Kim, A. Ankit, and K. Roy, "Geniex: A generalized approach to emulating non-ideality in memristive xbars using neural networks," *arXiv preprint arXiv:2003.06902*, 2020.

[91] C. Liu, M. Hu, J. P. Strachan, and H. Li, "Rescuing memristor-based neuromorphic design with high defects," in *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*, IEEE, 2017, pp. 1–6.

[92] I. Chakraborty, D. Roy, and K. Roy, "Technology aware training in memristive neuromorphic systems for nonideal synaptic crossbars," *IEEE Transactions on Emerging Topics in Computational Intelligence*, vol. 2, no. 5, pp. 335–344, 2018.

[93] I. J. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and harnessing adversarial examples," *arXiv preprint arXiv:1412.6572*, 2014.

[94] G. S. Dhillon, K. Azizzadenesheli, Z. C. Lipton, J. Bernstein, J. Kossaifi, A. Khanna, and A. Anandkumar, "Stochastic activation pruning for robust adversarial defense," *arXiv preprint arXiv:1803.01442*, 2018.

[95] J. Buckman, A. Roy, C. Raffel, and I. Goodfellow, "Thermometer encoding: One hot way to resist adversarial examples," in *International Conference on Learning Representations*, 2018. [Online]. Available: https://openreview.net/forum?id=S18Su--CW.

[96] A. Athalye, N. Carlini, and D. Wagner, "Obfuscated gradients give a false sense of security: Circumventing defenses to adversarial examples," *arXiv preprint arXiv:1802.00420*, 2018.

[97] D. Niu, Y. Chen, C. Xu, and Y. Xie, "Impact of process variations on emerging memristor," in *Proceedings of the 47th Design Automation Conference*, 2010, pp. 877–882.

[98] G. W. Burr, R. M. Shelby, S. Sidler, C. Di Nolfo, J. Jang, I. Boybat, R. S. Shenoy, P. Narayanan, K. Virwani, E. U. Giacometti, *et al.*, "Experimental demonstration and tolerancing of a large-scale neural network (165 000 synapses) using phase-change memory as the synaptic weight element," *IEEE Transactions on Electron Devices*, vol. 62, no. 11, pp. 3498–3507, 2015.

[99] S. Ambrogio, P. Narayanan, H. Tsai, R. M. Shelby, I. Boybat, C. Di Nolfo, S. Sidler, M. Giordano, M. Bodini, N. C. Farinha, *et al.*, "Equivalent-accuracy accelerated neural-network training using analogue memory," *Nature*, vol. 558, no. 7708, pp. 60–67, 2018.

[100] F. Cai, J. M. Correll, S. H. Lee, Y. Lim, V. Bothra, Z. Zhang, M. P. Flynn, and W. D. Lu, "A fully integrated reprogrammable memristor–cmos system for efficient multiply–accumulate operations," *Nature Electronics*, vol. 2, no. 7, pp. 290–299, 2019.

[101] M. Hu, J. P. Strachan, Z. Li, E. M. Grafals, N. Davila, C. Graves, S. Lam, N. Ge, J. J. Yang, and R. S. Williams, "Dot-product engine for neuromorphic computing: Programming 1t1m crossbar to accelerate matrix-vector multiplication," in *Design Automation Conference (DAC), 2016 53nd ACM/EDAC/IEEE*, IEEE, 2016, pp. 1–6.

[102] N. Papernot, P. McDaniel, I. Goodfellow, S. Jha, Z. B. Celik, and A. Swami, "Practical black-box attacks against machine learning," in *Proceedings of the 2017 ACM on Asia conference on computer and communications security*, 2017, pp. 506–519.

[103] C. Xie, J. Wang, Z. Zhang, Z. Ren, and A. Yuille, "Mitigating adversarial effects through randomization," *arXiv preprint arXiv:1711.01991*, 2017.

[104] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE conference on computer vision and pattern recognition*, Ieee, 2009, pp. 248–255.

[105] A. Madry, A. Makelov, L. Schmidt, D. Tsipras, and A. Vladu, "Towards deep learning models resistant to adversarial attacks," *arXiv preprint arXiv:1706.06083*, 2017.

[106] M. Andriushchenko, F. Croce, N. Flammarion, and M. Hein, "Square attack: A query-efficient black-box adversarial attack via random search," 2020.

[107] M. Schumer and K. Steiglitz, "Adaptive step size random search," *IEEE Transactions on Automatic Control*, vol. 13, no. 3, pp. 270–276, 1968.

[108] S. Moon, G. An, and H. O. Song, "Parsimonious black-box adversarial attacks via efficient combinatorial optimization," in *International Conference on Machine Learning (ICML)*, 2019.

[109] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds., Curran Associates, Inc., 2019, pp. 8024–8035.

[110] J. Hang, K. Han, H. Chen, and Y. Li, "Ensemble adversarial black-box attacks against deep learning systems," *Pattern Recognition*, vol. 101, p. 107 184, 2020.

[111] C. Guo, M. Rana, M. Cisse, and L. Van Der Maaten, "Countering adversarial images using input transformations," *arXiv preprint arXiv:1711.00117*, 2017.

[112] A. Madry, A. Makelov, L. Schmidt, D. Tsipras, and A. Vladu, "Towards deep learning models resistant to adversarial attacks," in *International Conference on Learning Representations*, 2018.

[113] A. Cappelli, R. Ohana, J. Launay, L. Meunier, I. Poli, and F. Krzakala, "Adversarial robustness by design through analog computing and synthetic gradients," *arXiv preprint arXiv:2101.02115*, 2021.

[114] D. Roy, I. Chakraborty, T. Ibrayev, and K. Roy, "Robustness hidden in plain sight: Can analog computing defend against adversarial attacks?" *arXiv preprint arXiv:2008.12016*, 2020.

[115] X. Guan, S. Yu, and H.-S. P. Wong, "A spice compact model of metal oxide resistive switching memory with variations," *IEEE electron device letters*, vol. 33, no. 10, pp. 1405–1407, 2012.

[116] S. Arora, R. Ge, B. Neyshabur, and Y. Zhang, "Stronger generalization bounds for deep nets via a compression approach," in *International Conference on Machine Learning*, PMLR, 2018, pp. 254–263.

[117] E. Duesterwald, A. Murthi, G. Venkataraman, M. Sinn, and D. Vijaykeerthy, "Exploring the hyperparameter landscape of adversarial robustness," *arXiv preprint arXiv:1905.03837*, 2019.

[118] C. Liu, M. Salzmann, T. Lin, R. Tomioka, and S. Süsstrunk, "On the loss landscape of adversarial training: Identifying challenges and how to overcome them," *Advances in Neural Information Processing Systems*, vol. 33, 2020.

[119] J. Kubilius, M. Schrimpf, H. Hong, N. J. Majaj, R. Rajalingham, E. B. Issa, K. Kar, P. Bashivan, J. Prescott-Roy, K. Schmidt, A. Nayebi, D. Bear, D. L. K. Yamins, and J. J. DiCarlo, "Brain-Like Object Recognition with High-Performing Shallow Recurrent ANNs," in *Neural Information Processing Systems (NeurIPS)*, H. Wallach, H. Larochelle, A. Beygelzimer, F. D'Alché-Buc, E. Fox, and R. Garnett, Eds., Curran Associates, Inc., 2019, pp. 12 785–12 796. [Online]. Available: http://papers.nips.cc/paper/9441-brain-like-object-recognition-with-high-performing-shallow-recurrent-anns.

[120] J. Dapello, T. Marques, M. Schrimpf, F. Geiger, D. D. Cox, and J. J. DiCarlo, "Simulating a primary visual cortex at the front of cnns improves robustness to image perturbations," *BioRxiv*, 2020.

A. TREE-CNN

A.1 Incremental CIFAR-100 Dataset

The 100 classes of CIFAR-100 were randomly arranged and divided in 10 batches, each containing 10 classes. We randomly shuffled numbers 1 to 100 in 10 groups and then used that to group classes. We list the batches in the order they were added to the Tree-CNN for the incremental learning task below.

0 chair, bridge, girl, kangaroo, lawn mower, possum, otter, poppy, sweet pepper, bicycle
1 lion, man, palm tree, tank, willow tree, bowl, mountain, hamster, chimpanzee, cloud
2 plain, leopard, castle, bee, raccoon, bus, rabbit, train, worm, ray
3 table, aquarium fish, couch, caterpillar, whale, sunflower, trout, butterfly, shrew, house
4 bottle, orange, dinosaur, beaver, bed, snail, flatfish, shark, tractor, apple
5 woman, fox, lobster, skunk, can, turtle, cockroach, dolphin, bear, pickup truck
6 lizard, road, porcupine, mouse, seal, sea, tiger, telephone, rocket, tulip
7 baby, motorcycle, elephant, clock, maple tree, mushroom, pear, orchid, spider, oak tree
8 wardrobe, squirrel, crocodile, wolf, plate, skyscraper, keyboard, beetle, streetcar, crab
9 snake, lamp, camel, pine tree, cattle, boy, rose, forest, television, cup

A.2 Final Tree-CNN for max children 5, 10, 20 (CIFAR-100)

We trained the *Tree-CNN* with the incremental CIFAR-100 dataset, and we set the maximum number of children a branch node can have as 5, 10, and 20. The corresponding 3 *Tree-CNN*s were labeled - *Tree-CNN-5*, *Tree-CNN-10*, *Tree-CNN-20*. The 2-level hierarchical structure of these *Tree-CNNs* after 9 incremental learning stages is shown in Fig. A.1 - A.3. The nodes marked 'yellow' indicate completely filled branch nodes (B), the ones marked 'blue' indicate branch nodes (B) that are partially filled, while those marked 'green' refer to leaf nodes (L).

ROOT						
	B	APPLE	BOWL	SWEET PEPPER	TABLE	WORM
	B	AQ. FISH	RAY	SHREW	TROUT	WHALE
	B	BABY	LIZARD	MUSHROOM	SPIDER	WOLF
	B	BEAR	BEAVER	MOUSE	PORCUPINE	SKUNK
	B	BED	BUS	COUCH	HOUSE	TRACTOR
	B	BEE	BUTTERFLY	CATERPILLAR	POPPY	SUNFLOWER
	B	BOTTLE	CAN	CLOCK	TELEPHONE	WARDROBE
	B	BRIDGE	CASTLE	PALM TREE	TANK	TRAIN
	B	CHIMPANZEE	CLOUD	MOUNTAIN	OTTER	WILLOW TREE
	B	CUP	KEYBOARD	LAMP	PLATE	TELEVISION
	B	DINOSAUR	ELEPHANT	FOX	KANGAROO	TIGER
	B	DOLPHIN	ROCKET	SEAL	SHARK	TURTLE
	B	FLATFISH	GIRL	HAMSTER	MAN	WOMAN
	B	LEOPARD	LION	POSSUM	RABBIT	RACCOON
	B	MAPLE TREE	OAK TREE	PLAIN	ROAD	SEA
	B	BOY	CAMEL	CATTLE	SQUIRREL	
	B	LOBSTER	ORCHID	ROSE	TULIP	
	B	BEETLE	COCKROACH	SNAIL		
	B	LAWN MOWER	PICKUP TRUCK	STREETCAR		
	B	BICYCLE	MOTORCYCLE			
	B	CRAB	SNAKE			
	B	CROCODILE	FOREST			
	B	ORANGE	PEAR			
	L	CHAIR				
	L	PINE TREE				
	L	SKYSCRAPER				

Figure A.1. : Tree-CNN-5: After 9 incremental learning stages

ROOT	B	AQ. FISH	BEETLE	BUTTERFLY	COCKROACH	CRAB
		ORCHID	POPPY	SPIDER	SUNFLOWER	TULIP
	B	BEAR	BOTTLE	CAN	CATERPILLAR	FOX
		PORCUPINE	ROCKET	SKUNK	TELEPHONE	TIGER
	B	BEAVER	DINOSAUR	FLATFISH	HAMSTER	KANGAROO
		POSSUM	SHARK	SHREW	SNAIL	WHALE
	B	BED	BICYCLE	CHAIR	COUCH	HOUSE
		MOTORCYCLE	PICKUP TRUCK	ROAD	TABLE	TRACTOR
	B	BEE	BOWL	CHIMPANZEE	LION	OTTER
		RABBIT	RACCOON	RAY	VLOUD	WORM
	B	BRIDGE	BUS	CASTLE	LEOPARD	MOUNTAIN
		PALM TREE	PLAIN	TANK	TRAIN	WILLOW TREE
	B	DOLPHIN	ELEPHANT	LIZARD	MAPLE TREE	MOUSE
		MUSHROOM	SEA	SEAL	TROUT	TURTLE
	B	CLOCK	CUP	LAMP	PLATE	SNAKE
		TELEVISION	WARDROBE			
	B	BABY	BOY	GIRL	MAN	WOMAN
	B	APPLE	ORANGE	PEAR	ROSE	SWEET PEPPER
	B	FOREST	OAK TREE	PINE TREE	STREET CAR	
	B	CAMEL	CATTLE	SQUIRREL		
	L	CROCODILE				
	L	LAWN MOWER				
	L	LOBSTER				
	L	SKYSCRAPER				
	L	WOLF				

Figure A.2. : Tree-CNN-10: After 9 incremental learning stages

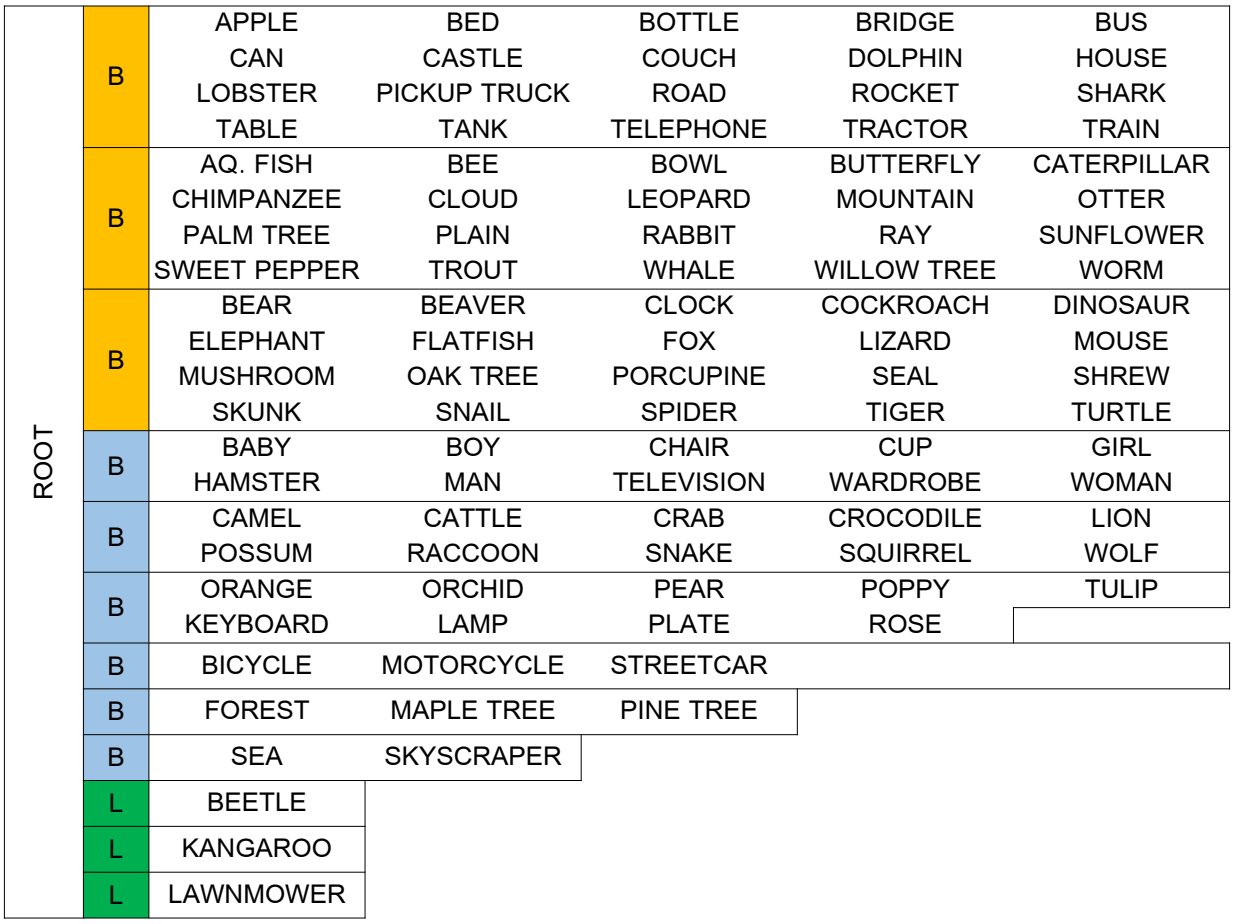


Figure A.3. : Tree-CNN-20: After 9 incremental learning stages

A.3 Full Simulation Results

Table A.1. : Normalized Training Effort as classes are added incrementally in batches of 10 (CIFAR-100)

Number of Classes	B:I	B:II	B:III	B:IV	B:V	Tree-CNN-5	Tree-CNN-10	Tree-CNN-20
20	0.08	0.17	0.19	0.20	0.20	0.18	0.18	0.18
30	0.12	0.25	0.29	0.30	0.30	0.26	0.26	0.27
40	0.16	0.34	0.38	0.39	0.40	0.34	0.34	0.35
50	0.20	0.42	0.48	0.49	0.50	0.42	0.43	0.42
60	0.24	0.51	0.58	0.59	0.60	0.50	0.51	0.52
70	0.28	0.60	0.67	0.69	0.70	0.59	0.60	0.60
80	0.33	0.68	0.77	0.79	0.80	0.67	0.68	0.68
90	0.37	0.77	0.87	0.89	0.90	0.74	0.74	0.75
100	0.41	0.86	0.97	1.00	1.00	0.82	0.83	0.84

Table A.2. : Test Accuracy as classes are added incrementally in batches of 10 (CIFAR-100)

Number of Classes	B:I	B:II	B:III	B:IV	B:V	Tree-CNN-5	Tree-CNN-10	Tree-CNN-20
20	64.30	73.60	77.40	78.50	81.10	77.80	81.35	78.35
30	52.47	67.17	72.27	77.57	79.30	72.70	77.00	74.57
40	46.68	61.72	67.55	72.08	74.35	72.15	72.90	70.75
50	41.42	58.74	64.74	69.62	71.82	69.28	67.40	67.42
60	37.98	55.80	61.85	66.95	69.57	67.42	65.73	64.40
70	35.43	52.96	59.10	64.23	67.21	65.13	62.91	62.07
80	34.55	51.68	57.77	61.81	66.03	64.07	61.73	61.60
90	33.88	51.09	55.77	62.21	64.90	63.52	60.93	60.88
100	31.73	48.68	54.16	60.48	63.05	61.57	60.46	59.99

B. STOCHASTIC NEURAL NETWORKS: ENERGY ESTIMATIONS

B.1 Energy Estimate

For any given convolutional layer, there are I input channels and O output channels. Let the size of the input be $N \times N$, size of the kernel be $k \times k$ and size of the output be $M \times M$. The number of memory access and computations in such a convolutional layer can then be formulated as listed in Table B.1.

Table B.1. : Operations in neural networks

Operation	Number of Operations
Input Read	$N^2 \times I$
Weight Read	$k^2 \times I \times O$
Computations (MAC)	$M^2 \times I \times k^2 \times O$
Memory Write	$M^2 \times O$

The number of memory-accesses N_{M-FP} and computations N_{C-FP} in a network with full-precision activations and weights are then given by:

$$\begin{aligned} N_{M-FP} &= N^2 \times I + k^2 \times I \times O + M^2 \times O \\ N_{C-FP} &= M^2 \times I \times k^2 \times O \end{aligned} \tag{B.1}$$

The number of memory-accesses N_{M-B} and computations N_{C-B} in a network with binary activations, are given as:

$$\begin{aligned} N_{M-B} &= N^2 \times I + k^2 \times I \times O + M^2 \times O \\ N_{C-B} &= M^2 \times I \times k^2 \times O \end{aligned} \tag{B.2}$$

Note that N_{M-FP} refers to full-precision memory access and N_{M-B} are binary memory accesses. In addition, N_{C-FP} refers to the number of FP MAC operations whereas N_{C-B} are the number of FP additions.

Hence, the energy consumed by a layer with 32-bit weights and 1-bit activations (E_B) is given by,

$$E_B = N_{M-B}E_{A-1} + N_{C-B}E_{AD-F} \tag{B.3}$$

Similarly, the energy consumed by a layer with 32-bit weights and 32-bit activations (E_{FP}) is computed as,

$$E_{FP} = N_{M-FP}E_{A-32} + N_{C-}E_{M-F} \tag{B.4}$$

VITA

Deboleena Roy received her B.Tech and M.Tech (Dual Degree) from Indian Institute of Technology (IIT), Kharagpur, India, in 2014. She joined Qualcomm Bangalore Design Centre in July 2014 as a Front End Design Engineer. She was part of the Design for Power team and worked on making Value Tier Snapdragon chipsets more power-efficient and market competitive. She joined Purdue University in Fall 2016 and is currently pursuing PhD degree under the guidance of Prof. Kaushik Roy. She was an intern at Samsung Research, Plano, TX in Summer 2019 where she worked on re-lighting multi-frame images. In summer 2020, she was an intern at Facebook, New York, NY, and she worked on designing machine learning models to predict if a webpage is a news article. Her primary research area is development and implementation of neuro-inspired algorithms for cognitive applications such as perception, reasoning and decision making.

PUBLICATIONS

1. **Deboleena Roy**, Priyadarshini Panda, and Kaushik Roy. "Tree-CNN: A Hierarchical Deep Convolutional Neural Network for Incremental Learning" *Elseiver Neural Networks*, 2018
2. **Deboleena Roy**, Priyadarshini Panda, Kaushik Roy. "Synthesizing Images from Spatio-Temporal Representations using Spike-based Backpropagation", *Frontiers in Neuroscience*, 2019
3. Abhronil Sengupta, Gopalakrishnan Srinivasan, **Deboleena Roy**, Kaushik Roy. "Stochastic Inference and Learning Enabled by Magnetic Tunnel Junctions", *IEEE International Electron Device Meeting (IEDM)*, 2018 (All authors contributed equally)
4. **Deboleena Roy**, Indranil Chakraborty, and Kaushik Roy. "Scaling Deep Spiking Neural Networks with Binary Stochastic Activations", *IEEE International Conference on Cognitive Computing (ICCC)*, 2019.
5. **Deboleena Roy**, Indranil Chakraborty, Timur Ibrayev, and Kauhik Roy. "On the Intrinsic Robustness of NVM Crossbars Against Adversarial Attacks", *To appear in 58th ACME/IEEE Design Automation Conference (DAC)*, 2021
6. **Deboleena Roy**, Chun Tao, Indranil Chakraborty, and Kaushik Roy. "On the Noise Stability and Robustness of Adversarially Trained Networks on NVM Crossbars", Under review in *40th IEEE/ACM International Conference on Computer Aided Design*, 2021