EFFICIENT NEURAL NETWORK PROCESSING VIA MODEL COMPRESSION AND LOW-POWER FUNCTIONAL UNITS

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF ENGINEERING AND SCIENCE
OF BILKENT UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR
THE DEGREE OF
MASTER OF SCIENCE
IN
ELECTRICAL AND ELECTRONICS ENGINEERING

By
Ali Necat Karakuloğlu
December 2024

Efficient Neural Network Processing via Model Compression and Low-Power Functional Units
By Ali Necat Karakuloğlu
December 2024

We certify that we have read this thesis and that in our opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Emine Ülkü Sarıtaş Çukur(Advisor)

Burçin Çakır(Co-Advisor)

Alper Demir

Tolga Çukur

Approved for the Graduate School of Engineering and Science:

Orhan Arıkan
Director of the Graduate School

ABSTRACT

EFFICIENT NEURAL NETWORK PROCESSING VIA MODEL COMPRESSION AND LOW-POWER FUNCTIONAL UNITS

Ali Necat Karakuloğlu
M.S. in Electrical And Electronics Engineering
Advisor: Emine Ülkü Sarıtaş Çukur
Co-Advisor: Burçin Çakır
December 2024

We present a framework that contributes neural network optimization through novel methods in pruning, quantization, and arithmetic unit design for resource-constrained devices to datacenters. The first component is a pruning method that employs an importance metric to measure and selectively eliminate less critical neurons and weights, achieving high compression rates up to 99.9% without sacrificing significant accuracy. This idea is improved by a novel pruning schedule that optimizes the balance between compression and model's generalization capability. Next, we introduce a quantization method that combines with pruning to improve hardware compatibility for floating point format, offering efficient model compression and fast computation and general usability. Finally, we propose a logarithmic arithmetic unit that designed as an energy-efficient alternative to conventional floating-point operations, providing precise and configurable processing without relying on bulky lookup tables. Extensive evaluations across different datasets and CUDA-based simulations and Verilog based hardware designs indicate that our approaches outperforms existing methods, making it a powerful solution for deploying artificial intelligence models more efficiently.

Keywords: Efficient, Neural Networks, Pruning, Quantization, Approximate Computing.

ÖZET

MODEL SIKIŞTIRMA VE DÜŞÜK GÜÇ FONKSİYONEL ÜNITELER İLE VERİMLİ NÖRAL AĞ İŞLEME

Ali Necat Karakuloğlu
Elektrik Elektronik Mühendisliği, Yüksek Lisans
Tez Danışmanı: Emine Ülkü Sarıtaş Çukur
İkinci Tez Danışmanı: Burçin Çakır
Aralık 2024

Kaynak kısıtlı cihazlardan veri merkezlerine kadar geniş bir yelpazede sinir ağı optimizasyonuna katkı sağlayan, budama, nicemleme ve aritmetik birim tasarımı konularında yenilikçi yöntemler sunan bir çerçeve sunuyoruz. İlk bileşen, önemsiz nöronları ve ağırlıkları seçici olarak ölçüp ortadan kaldıran bir önem metriği kullanan bir budama yöntemidir ve bu sayede % 99.9'a varan yüksek sıkıştırma oranlarına önemli bir doğruluk kaybı olmadan ulaşılmaktadır. Bu fikir, sıkıştırma ve modelin genelleme yeteneği arasındaki dengeyi optimize eden yenilikçi bir budama planıyla daha da geliştirilmiştir. İkinci olarak, budama ile birleşerek donanım uyumluluğunu artıran, verimli model sıkıştırma, hızlı hesaplama ve genel kullanılabilirlik sunan bir nicemleme yöntemi sunuyoruz. Son olarak, geleneksel kayan nokta işlemlerine enerji verimli bir alternatif olarak tasarlanan, büyük tablo arama yapılarına ihtiyaç duymadan hassas ve yapılandırılabilir işlem sunan bir logaritmik aritmetik birim öneriyoruz. Farklı veri setleri, CUDA tabanlı simülasyonlar ve Verilog tabanlı donanım tasarımları üzerinde yapılan kapsamlı değerlendirmeler, önerilen yöntemlerimizin mevcut yöntemlerden üstün olduğunu ve yapay zeka modellerinin daha verimli bir şekilde çalışması için güçlü bir çözüm sunduğunu göstermektedir.

Anahtar sözcükler: Verimli, Sinir Ağı, Budama, Nicemleme, Yaklaşık Hesaplama.

Acknowledgement

I wish to share my greatest gratitude to my thesis advisor, Dr. Burçin Çakır, for her patience, guidance, understanding, and, most importantly, her undying support throughout our studies and this work. Her mentorship has been a source of inspiration and encouragement, motivating me to pursue and complete this research.

I extend my heartfelt thanks to Assoc. Prof. Emine Ülkü Sarıtaş Çukur for her understanding, and support, which have been invaluable. Similarly, I would like to express my sincere appreciation to the thesis committee members, Prof. Alper Demir and Prof. Tolga Çukur, for their feedback and contributions to this work.

I would also like to acknowledge that the numerical calculations reported in this thesis were partially performed at TÜBİTAK ULAKBİM, High Performance and Grid Computing Center (TRUBA resources). I am grateful for the infrastructure and support provided, which were essential for conducting this research.

I want to extend my heartfelt thanks to Aydan Hanım and Funda Hanım for their support, patience, and help. They were always warm, understanding, and ready to assist with my questions, and their kindness has meant so much to me throughout this process.

I want to express my thanks to my friends for our valuable share of thoughts. Finally, I would like to extend my deepest gratitude for my mother and father for their unlimited support, understanding and love.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Deep learning, as an emerging field and a huge economy, is effecting our lives more and more each day. With this substantial impact, the amount of data and models that are designed to process and learn from this large volume of information are also growing with a similar rate. Consequently, computational demand to process this massive amount of data is increasing. This brings about energy and processing concerns for both battery-limited platforms and energy-intensive high performance data/computation centers. The need for increased computational power also becomes crucial in time-sensitive applications such as autonomous driving and object detection, where model runtime complexity directly affects performance. As a result, the necessity of designing power- and cost-efficient solutions emerges as a significant research area.

To address economical, environmental, and time-related constraints, many researchers have proposed methods on reducing the computational load of deep learning applications. Pruning and quantization are two main areas where research effort is concentrated to reduce the computational burden of neural networks. Another promising field is approximate computing, which can help further improve the overall cost of neural network training and inference.

Neural network pruning is based on elimination of redundant parameters that

are not effective in model's generalization capability. Crucial point in neural network pruning algorithms is to determine which parameters to prune. Since large models offer a very large search space (for finding significance of each weight), researchers have been developing methods to find unimportant parameters and eliminate them. Another important field, quantization focuses on reducing the number of bits to represent a parameter to reduce memory usage during inference and training. Studies on quantization mainly focuses on creating new numerical representations to replace high precision floating point representation. Both pruning and quantization reduces model size, resulting in improvements on memory requirements. Pruning has benefits for general purpose computing devices like Central Processing Units (CPUs) and Graphics Processing Units GPUs. Quantization on the other hand, usually requires hardware modifications to work effectively.

In addition to pruning and quantization, numerous approximate computing approaches have also been proposed for neural network inference and training, as floating point units are usually hardware costly, power hungry with redundant precision overheads. Integer arithmetic and logarithm based solutions are given as a promising alternative to floating point units.

In the first chapter of this thesis, we share our work on pruning neural networks to reduce model size and computational requirements. In the heart of this effort, the pruning mechanism ranks individual neurons (or channels) and regulate the model by suppressing low ranked channels. Neurons (or channels) are ranked by measuring the difference between perturbed and original outputs of a layer. Next, channels are suppressed or boosted in correlation with these ranks. Suppression makes weak channels less effective on the model output and gives priority to better channels. This results in easier pruning process since weak channels gets weaker by suppression and finally mostly pruned. To increase the effectiveness of our pruning method, we also examine the effect of pruning schedule. Pruning schedule is the control mechanism that determines sparsity level of a model at a given level. We investigate other pruning schedules and show the effectiveness of our pruning schedule, in terms of generalization capability, in comparison to other studies. The results show that our pruning schedule achieves higher accuracy at

same sparsity levels.

In the next chapter, our work extends pruning and regularization mechanism by introducing hardware-compatible floating-point quantization. Quantization uses the channel rankings to adjust the precision of parameters within each neuron. Specifically, we reduce the floating-point precision of weights in lower-ranked neurons by eliminating mantissa bits starting from the least significant bits (bit-adjustment). Bit-adjustment technique, combined with pruning, significantly reduces the number of unique weights within a model. Our observations show that with pruning and bit adjustment, the number of unique parameters reduces significantly (ranging from 40 to 200), making the model highly compressible.

In the final chapter, we propose a logarithmic arithmetic unit capable of delivering equal or near-floating-point accuracy with simpler hardware and lower memory costs. We present an approximate computing unit based on a novel, adjustable, lightweight and precise enough logarithm based arithmetic. Our hardware design provides a configurable solution, allowing for easy parameter tuning across various workloads in the hardware and enabling optimal performance for different model architectures and datasets. Our experiments show that the proposed logarithmic design is a prominent alternative to floating based arithmetic in terms of precision, model accuracy, hardware cost and power consumption.

In conclusion, this thesis provides hardware-software co-design for efficient processing of neural networks by studying and developing solutions based on pruning, quantization and approximate computing.

Chapter 2

Trainable Neuron Ranking for Neural Network Regularization and Pruning

2.1 Introduction

The ever-growing complexity of neural network architectures and datasets has led to an unsustainable surge in computational costs and energy consumption. In response to the growing need for efficient neural network training, considerable research efforts have been directed towards this field. Neural network pruning has emerged as a key research area within this domain, aiming to reduce both computation and memory requirements. Sparse models, achieved through pruning, necessitate fewer memory read/write operations. This translates to lower power consumption due to the ability to skip multiplications by zero, provided that hardware supports such optimizations.

A range of methods have been developed to improve the efficiency of training and inference in deep neural networks. These techniques aim to reduce the network's complexity by removing redundant or unimportant connections (weights)

and neurons. However, this raises the question: what makes an ideal pruning algorithm? In theory, a perfect pruning algorithm would achieve significant reductions in network size (number of weights) with minimal effort, while maintaining the model's generalization ability on the target dataset. This ideal algorithm would essentially identify the minimum set of weights necessary for accurate performance. Unfortunately, achieving this ideal is challenging due to the complex nature of neural networks and the difficulty of precisely pinpointing the least important weights without impacting accuracy. This would be computationally infeasible for models with millions of parameters, as the search space grows exponentially with the number of weights ($2^{totalweights}$).

Identifying unimportant weights-connections in a neural network is the first challenge that researchers have focused on. One influential work by LeCun et al. [1] introduced an approximation of weight salience based on the Hessian of the loss function. This approach helps identify weights that contribute less to the overall gradient and are therefore potentially less important. Another approach, proposed by Molchanov et al. [2], utilizes Taylor expansion to approximate the effect of weight pruning on the model's output. This method, along with its computationally efficient improvement by Molchanov et al. [3], represents a gradient-based approach to weight importance estimation. The work by Lee et al. [4] estimate the impact of an indicator function (pruning decision) that prunes the model by computing the loss with respect to each weight before training. They prune connection at the beginning of training process using this connection sensitivity estimation.

After identifying weights or connections for removal, the next step requires determining how these are reflected in the network pruning granularity, where we have two main categories: Magnitude based unstructured pruning and structured pruning.

Unstructured pruning algorithms mainly focuses on the magnitude (absolute value) of weights to determine their importance. A common approach, introduced by Han et al. [5], involves sorting weights by their magnitudes and eliminating the weights below a threshold. This method is efficient but can potentially remove

important weights with small magnitudes. Building on this idea, works by Evci et al. [6] and Zhu and Gupta [7] proposed layer-wise magnitude-based pruning. Instead of pruning uniformly across the entire network, they consider the importance of weights within each layer. This approach can lead to more efficient pruning by focusing on less critical layers. Another advancement in magnitude-based pruning involves weight normalization methods, as proposed by Lee et al. [8]. These methods normalize weight magnitudes before pruning, preventing a layer from being entirely removed and mitigating the influence of large magnitude discrepancies between layers. However, a common downside for all of the methods above is that they do not acknowledge activation importance. This creates a gap between training and pruning processes.

Structured pruning methods are based on removing entire channels or neurons (we will refer both of them as neurons) from the network. This approach can lead to a more structured sparsity pattern, potentially benefiting hardware acceleration on parallel computing platforms. Works by Liu et al. [9], Zhuang et al. [10], and You et al. [11] exemplify neuron pruning techniques. You et al. [11] use a gate system to control filter impact on the output. They scale the outputs of the neurons based on this system while prune unimportant neurons. Similarly, Liu et al. [9] train a large network then removes neuron and slims the network based on the scaling factor of each neuron in batch normalization layers. Zhuang et al. [10] proposes a mechanism that forces neurons towards either an active or inactive state for more targeted neuron pruning compared to traditional methods. As a recent work, Sun et al. [12] focuses on pruning Large Language Models (LLMs) using l_2-norm of activations and creating saliency with the corresponding l_2-norm and weight magnitude. Concisely, structured pruning methods reduce the achievable compression since, the granularity does not allow further pruning of connections connected to a neuron. In addition, coarse pruning degrades accuracy much more than removal of an individual weight.

Another difficulty with pruning algorithms is deciding sparsity level. A common way to decide sparsity level is creating a sparsity schedule. Existing pruning schedules face a trade-off between achieving sparsity (weight removal) and maintaining model stability during training. Constant pruning rate, implemented by

Han et al. [5], removes a significant portion of weights upfront without further training. This aggressive approach can significantly reduce the search space for the model, potentially hindering its ability to learn effectively. On the other hand, gradual pruning schedules, like the one proposed by Zhu and Gupta [7], introduce weights removal more slowly throughout training. While this approach mitigates the risk of instability, it can also lead to the model reaching a stable point with a larger number of weights than necessary.

In this work, we propose a novel pruning algorithm that combines the strengths of both structured and unstructured (magnitude) pruning with a new pruning schedule. This approach aims to achieve efficient weight removal while promoting model regularization during training. Unlike existing methods that focus solely on weights or neurons, our approach considers the resulting effect on layer activations, leading to more informed pruning decisions. We introduce a new pruning mechanism that leverages an automatically controlled neuron ranking method to estimate the impact of weight removal on each neuron. This ranking guides the pruning process, ensuring efficient weight removal while promoting model regularization during training. To estimate the impact of weight removal, we use a simple yet effective weight disturbance scheme. Instead of directly removing weights, we temporarily disturb them proportional to their magnitudes. This approach avoids introducing excessive noise that could invalidate the measurement of output disturbance. To address the limitations regarding pruning scheduling, we propose an exponential decay pruning schedule, which balances the need for sparsity with training stability. We evaluate our strategy of pruning by training-pruning various models on CIFAR-10 and ImageNet dataset. We observe that our approach overperforms the other methods up to 13.15% over 99.9% pruning. To summarize, this work makes the following key contributions: ***(1)*** Introduces a novel method for estimating the importance of neurons in neural networks and a trainable ranking mechanism. ***(2)*** Develops a technique that regularizes the model by integrating pruning scores/decisions directly into the training process. ***(3)*** Employs a pruning schedule that preserves the model's search space, ensuring effective training and pruning.

2.2 Method

Our framework consists of two parts for pruning and regularization stages. The algorithm start by disturbing weights by adding a computationally efficient, proportional noise to each value. Then, the percentage change of the outputs are measured by the ratio of disturbed and original output. Utilizing this measurement and a scoring function, the neurons within a layer are sorted. Based on this rank, the model is pruned by creating saliency for the weights corresponding to a neuron (or channel). To rank the neurons, we use a function that penalizes the ratios closer to 1. Ranking is used for creating a semi-structured pruning algorithm and regularizing the neurons by boosting or suppressing the weights during training process. Figure 2.1 illustrates the neuron ranking and pruning approach. The figure summarizes that weights connected to weak neurons are more probable to be pruned because of the low ranking in the corresponding layer.

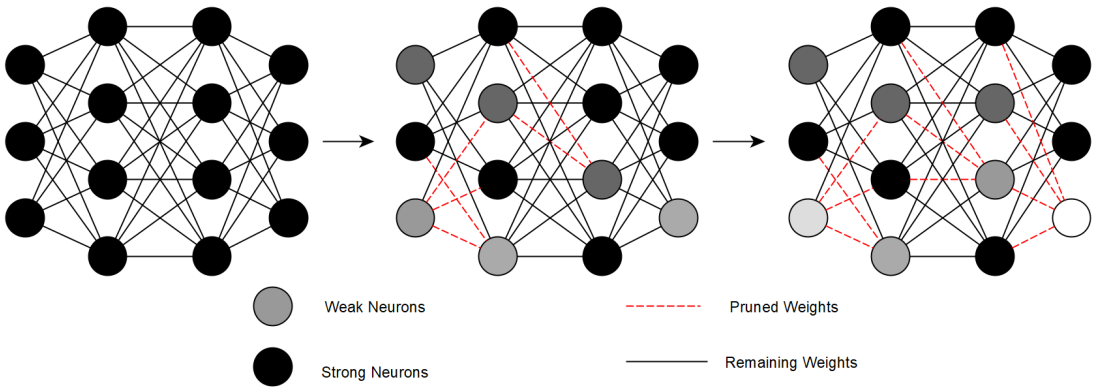


Figure 2.1: Neuron ranking and pruning

2.2.1 Weight Disturbance and Noise Modelling

Given that the initial weight representation (w) is provided in floating point representation [13], the disturbance process is modelled as noise addition. In essence, we do not actually add noise but change the weight by removing its mantissa field. Since clearing mantissa field of a floating number does not change its exponent

and sign field, this has two advantages: First, the sign of the disturbed weight remains the same with the original one. Second, we make sure that weight is not changing abruptly in magnitude. Both of those properties prevent erroneous measurement of neurons response to the change. Equation 2.1 to 2.4 summarize the noise modeling, where w_d represents the disturbed output and δ denotes the noise.

$$w = sgn \cdot 2^{e-127} \cdot (1 + mantissa) \tag{2.1}$$

$$w_d = sgn \cdot 2^{e-127} \cdot 1 \tag{2.2}$$

$$w - w_d = sgn \cdot 2^{e-127} \cdot (mantissa) \tag{2.3}$$

$$\delta = w - w_d \tag{2.4}$$

Equations above represent weight disturbance process and resulting change in the weight magnitude. δ (noise) is computed as the difference between w and w_d. The sign bit does not change with the disturbance process.

2.2.2 Channel Ranking

First step to rank neurons is measuring the percentage change of the output at the next layer when mantissa bits are set to 0 (the proportional noise addition as explained above). This is achieved by dividing original output magnitude to the disturbed output magnitude. However, using this calculation directly as the merit is not practical. This ratio (R_{out}) consists of numbers close to 0 as well as very large values. It is not suitable to use it directly as a scoring or regularization metric. (Here, we note that ratios close to 0 and infinity are equally important as they both represent an important change at the output value.) Hence, to rank the neurons fairly for both cases, we pass the ratio through a soft-gating sigmoid (S_{out}, Equation 2.7) function centered around 1. The plot of the function is given in Figure 2.2. If the ratio is 1 or a close value to 1, function gives minimum output. On the other hand, if the ratio is close to 0 or a large value, result becomes 1. Thus, we say that ratios close to 1 are penalized by the function. Ratio being 1 means that the output does not change when the weights are disturbed. We use a parameter ϵ to configure aggressiveness of the ranking function (Equation 2.7).

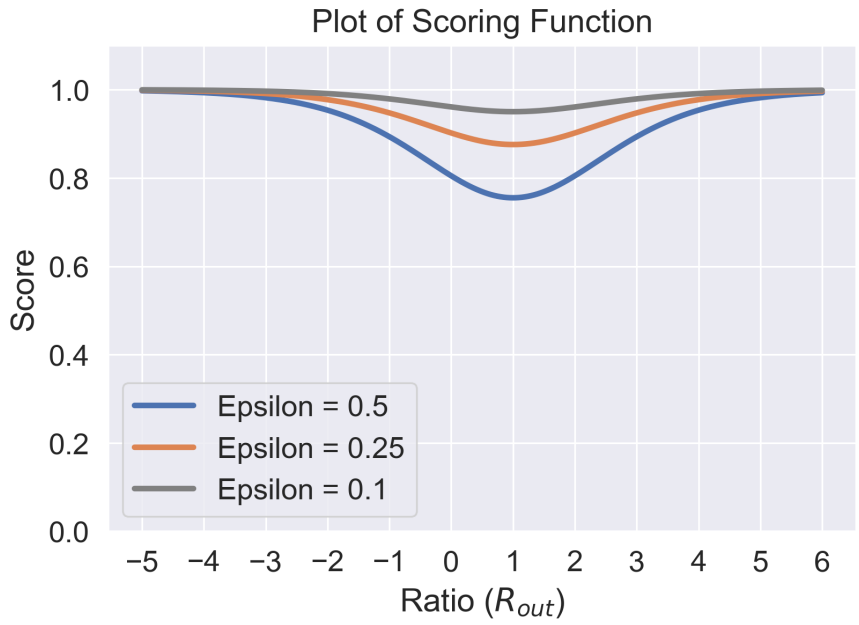


Figure 2.2: Grading of neurons based on the Ratio (R_{out}) with (ϵ) adjusting the aggressiveness of the ranking.

Comparing the output of disturbed weights and original weights allow us to measure how noise addition affects the layer output. If the output has changed drastically, we boost that channel because, it means that the channel has strong relevance with the input (i.e., it extracts relevant information from the input data for the next layer). Given an input x, weights w, and convolution (or linear) operation ($\otimes$) output $y^l(x^l; w^l)$, output ratio R_{out} and S_{out} are formulated as follows:

$$y^l(x^l; w^l) = x^l \otimes w^l \tag{2.5}$$

$$R_{out}(x^l; w^l, \delta) = \frac{\sum^N \sum^H \sum^W |\frac{y^l(x^l; w^l)}{y^l(x^l; w^l + \delta)}|}{NHW} \tag{2.6}$$

$$S_{out}(x^l; w^l, \delta) = \sigma(\text{-}R_{out} \text{ - } \epsilon + 1) + \sigma(R_{out} \text{ - } \epsilon \text{ - } 1) \tag{2.7}$$

where δ denotes noise, x represents input to layer l, w is layer weight, σ is the sigmoid function and (ϵ) is the trainable parameter that controls the aggressiveness of channel scoring, and for convolutional layers, N: batch, H: output height, W: width and, C: channel. In a convolutional neural network, the layer output $y \in \mathbb{R}^{NxHxWxC}$ has multiple dimensions. To reflect the information gathered in

$\frac{y^l(x^l;w^l)}{y^l(x^l;w^l+\delta)}$ for the channel C, the it is summed along N, H and W, resulting in C dimensional score vector.

To reflect the score calculation and ranking information in forward propagation as well, the final output of the layer ($f^l(x^l;w^l)$) is updated as the following:

$$z^l(x^l;w^l) = S_{out}(x^l;w^l,\delta)y^l(x^l;w^l) \tag{2.8}$$

$$f^l(x^l;w^l) = \theta(S_{out}(x^l;w^l,\delta)y^l(x^l;w^l) \tag{2.9}$$

where θ is the nonlinear activation function, $y^l(x^l;w^l)$ defines the convolved input and kernel for a CNN layer or matrix multiplication for a linear layer. This helps refining the model during training process.

Trainable Score Mechanism: As mentioned previously, the parameter epsilon (ϵ) controls the aggressiveness of our channel ranking mechanism. Larger ϵ values increase the score differences between channels, creating more pronounced boosting effect for high-scoring channels and stronger suppression of low-scoring ones. To achieve optimal ranking based on the model's performance, we make ϵ trainable. During backpropagation, the loss function (L) influences the update of epsilon (see Equation 2.10). When the loss is high, indicating a negative impact on model performance, the update reduces the value of epsilon, leading to a less aggressive ranking. This mechanism ensures that the ranking adapts to the model's needs, achieving optimal pruning decisions.

$$\frac{\partial L^l}{\partial \epsilon^l} = \frac{\partial L^l}{\partial f^l}\frac{\partial f^l}{\partial S^l}\frac{\partial S^l}{\partial \epsilon^l} \tag{2.10}$$

Discussion on Measuring Output Perturbance: Assuming weights are fixed and input is stochastic on the sample data, we can find the information added by each neuron to each corresponding output. The quality of this delivery can be assessed by measuring the output distribution difference between the perturbed output and the original output. This approach is grounded in information theory [14], where the divergence between probability distributions represents the

information loss or gain. However, directly approximating the probability distributions and calculating divergence, such as Kullback-Leibler divergence, is complex due to the high-dimensional nature of neural network outputs, the lack of analytical tractability and the computational complexity of approximating a distribution. To address this, we propose a data-driven approximation method to measure the shift in output distributions, which can be thought as a rough measure of divergence in the distributions.

In a neural network, the input data is inherently stochastic, varying from one sample to another, while the weights are trained to capture the patterns within the input distribution [15]. Weights gain relevance/evolve within the input context. This highlights the importance of evaluating the information transfer quality for a given set of weights.

Pruning Effect of Score Mechanism: Channel scoring allows us to prioritize pruning decisions of weights, with lower scores indicating less importance and a higher likelihood of being pruned within that channel.

By reducing the influence of gradients in low-scoring channels, score-based modulation helps the gradual suppression of less important channels during training (see Equation 2.11). This prevents abrupt changes in their weights, enabling them to be pruned more progressively. This process, where the model adapts to the score information (see Equation 2.9) and pruning decisions while training (explained in the following part, Section 2.2.3), leads to compression that is more aligned with the model's learning capability.

$$\frac{\partial L^l}{\partial w^l} = \frac{\partial L^l}{\partial z^l}\frac{\partial z^l}{\partial y^l}\frac{\partial y^l}{\partial w^l} \text{ where, } \frac{\partial z^l}{\partial y^l} = S_{out}(x^l; w^l, \delta) \tag{2.11}$$

Additionally, scoring influences how gradients propagate through the network. Channels (or neurons) with low scores experience smaller updates during backpropagation (see the expression for $\partial L^l/\partial x^l$ corresponding to the loss gradient of the previous layer $\partial L^{l-1}/\partial f^{l-1}$ in Equation 2.12, where the term $S_{out} = \partial z^l/\partial y^l$

imposes smaller updates on low scored neurons). This effectively creates a feedback mechanism where low-scoring units in a layer influence the updates received by their corresponding inputs in the previous layer. This modulation encourages the network to focus on high-scoring channels, helping faster convergence and improved generalization performance on validation data due to fading weights rather than sudden pruning updates.

$$\frac{\partial L^l}{\partial x^l} = \frac{\partial L^l}{\partial z^l} \frac{\partial z^l}{\partial y^l} \frac{\partial y^l}{\partial x^l} \tag{2.12}$$

Regularization Effect of Score Mechanism: Our scoring mechanism introduces two key benefits during training. First, it increases the variance of outputs within a layer during forward propagation due to redundancy removal of similar neurons (channels)/increased uniqueness in output values (leading to a more flattened distribution within each layer output). This new search space can provide the subsequent layer with a distilled set of inputs, potentially benefiting its learning ability by removing redundant parameters. Secondly, the scoring mechanism acts as a regularizer by penalizing channels with outputs that does not carry important information for the next layer. This encourages the model to focus on informative channels, promoting sparsity and potentially reducing overfitting due to compressed network architecture (increased generalization capacity). The focus on informative channels can be seen as a form of channel selection pressure driven by the scoring mechanism, leading to a more efficient network architecture through a process of 'channel evolution'.

Combination with Batch Normalization Layer: Batch Normalization (BN) by Ioffe [16] typically aims for unit variance and zero mean outputs to prevent abrupt change in the output. This prevents irregular input flow to the next layers. However, BN potentially conflicts with our neuron ranking mechanism. Neuron ranking methods are based on diversify layer outputs as mentioned by Liu et al. [9], He and Xiao [17]. Similarly, our scoring technique aim to increase the variability among activations, which can beneficial for learning and pruning since this approach combines pruning (or suppression) of neurons with training

process. As mentioned before, scoring applied on outputs creates a focus on more important neurons.

To address this conflict, we propose an approach that combines BN with scoring. Instead of scaling the layer's output directly with the score, we multiply the score with the BN weight of each channel. This approach allows us to incorporate the scoring information while preserving the desirable properties of BN. We skip normalization part in BN layer to combine activations with their ranks. This removes adversarial effect of normalization of activations. The new BN layer becomes as follows:

$$z = \frac{y - E[y]}{\sqrt{Var[y] + \epsilon_{BN}}} \cdot \gamma \cdot S_{out} + \beta \tag{2.13}$$

2.2.3 Weight Pruning

Scoring computation gives as a quantitative value to sort channel importance and propagate this on weights (a hierarchical ranking mechanism for weight pruning). Using this ranking, we impose importance on weights belonging to a neuron. We multiply the weight magnitudes with the score and generate a weight importance criterion (saliency). Then, to normalize this saliency we apply a method proposed by Lee et al. [8] (Layer Adaptive Magnitude Pruning (LAMP)), a magnitude normalization technique well-suited for global pruning strategies that efficiently normalizes weight magnitudes, allowing for effective saliency-based comparison across the entire layer.

To summarize, our pruning technique is structured as follows: ***(1)*** Create $S^t_{pruning}$ for a layer. ***(2)*** Generate weight ranking by multiplying channel score with the absolute value of the corresponding weights and apply a normalization strategy [8] to normalize *weight_score* values. ***(3)*** Based on the ranking, prune the $p(t)$ percent of the weights, where t is the training step.

$$S^t_{pruning} = \frac{S^{t-1}_{pruning} + S^t_{out}}{2} \tag{2.14}$$

$$weight\_score = S^t_{pruning} \cdot |weight| \tag{2.15}$$

To take into account the information generated by the previous training steps, we accumulate the current score calculation S^t_{out} using the above formulations. In Equation 2.14, $S^t_{pruning}$ and $S^{t-1}_{pruning}$ denote weighted-accumulated score of the layer at step t and $t-1$. Hence, the channel importance metric, $S^t_{pruning}$, used at $weight\_score$ calculation (Equation 2.15) represents the weighted-accumulation of previous scores calculated during training process.

2.2.4 Pruning Schedule

Pruning schedule is an important part of the pruning method that combines with weight importance metric to create complete pruning algorithm. Given a final sparsity (P_{final}) constraint, pruning schedule determines the percentage of the parameters to be pruned at each training iteration. In this chapter, we compare and explain two commonly studied methods constant and polynomial pruning rate with our proposed method based on exponential decay.

Exponential Decay: Our pruning schedule utilizes an exponential decay function governed by a configurable time constant τ. This time constant controls the rate of weight reduction, mimicking natural decay processes. It allows for aggressive pruning initially, significantly reducing the number of weights early in training. This rapid reduction narrows the search space efficiently, helps the model focus on essential weights. As training progresses, the pruning rate softens due to the exponential nature. This ensures sufficient number of remaining weights for fine-tuning towards the end. Compared to constant rate pruning, which removes a large portion of weights at the very beginning, and polynomial decay, which leads to a gradual decrease, our exponential schedule offers a balanced approach as can be seen from the Figure 2.3. It achieves significant early

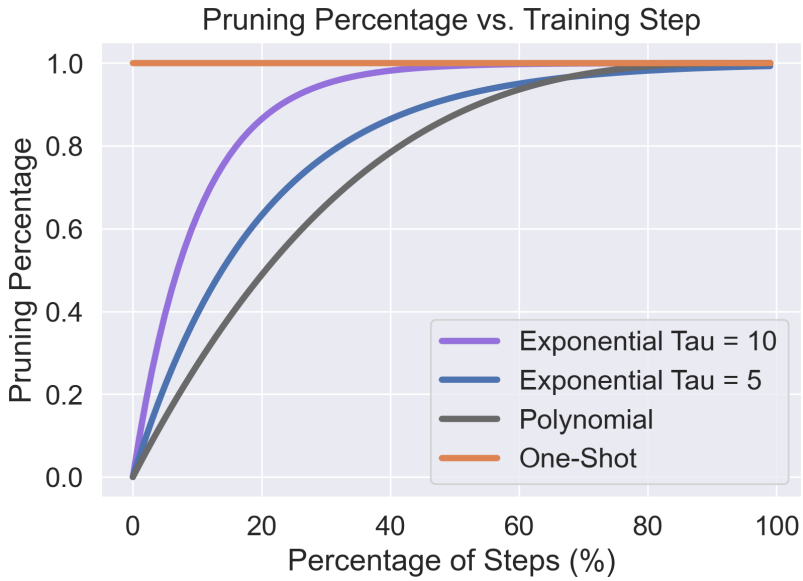


Figure 2.3: Pruning percentage as training progresses

reduction while leaving room for later refinement. Assuming model takes T steps to train, pruning percentage at step t with desired pruning percentage P_{final} is formulated as $P_{final} \cdot (1 - e^{-\tau \frac{t}{T}})$ for exponential pruning schedule. As parameter τ increases, exponential schedule becomes more aggressive and reaches to the final pruning in less number of training steps. Therefore, we choose a moderate τ value.

Table 2.1: Pruning schedule function table

Pruning Schedule	$p(t)$
Exponential Decay	$P_{final} \cdot (1 - e^{-\tau \frac{t}{T}})$
Gradual Decay	$P_{final} \cdot (1 - \frac{t}{T})^3$
Constant	P_{final}

Gradual Decay: Gradual decay function proposed by Zhu and Gupta [7] uses a polynomial function to control pruning percentage for a model given the final pruning ratio and total step sizes. Assuming the model takes a total of T steps to train, pruning percentage at a step t is $P_{final} \cdot (1 - \frac{t}{T})^3$. Compared to exponential pruning schedule, gradual decay schedule starts pruning softly and becomes more aggressive as t approaches to T. However, soft pruning at the beginning

of the training process makes model fixed with unnecessary parameters. Similarly, aggressive pruning towards the end of training creates a sudden drop in generalization capability of the model.

Constant Pruning: Constant rate pruning used by Han et al. [5] aggressively eliminates the parameters at the beginning of the training process. However, this makes model capacity too small too early. Removal of significant portion of weights shrinks the search space for the model and reduces the achievable final accuracy.

2.3 Experimental Results

To evaluate the effectiveness of our regularization-pruning technique (combining Layer-Adaptive Magnitude-based Pruning (LAMP) with our ranking method), we conducted experiments on various convolutional neural networks (CNNs) including VGG-16, VGG-16 with batch normalization [18], ResNet-18, ResNet-50 [19], EfficientNet-V2s [20], MobileNet-V2 [21, 22], and Inception-V3 [23]. We used ResNet-18 architecture and other models with CIFAR-10 dataset [24]. For ImageNet classification dataset [25], we used other models and ResNet-50 architecture to show the effect on larger architecture and dataset. To evaluate performance on transformer [26] architectures, we used implementation of GPT-2 [27] by Karpathy [28] finetuning on OpenWeb dataset [29]. All models were trained with consistent hyperparameters.

The results show that our method demonstrates clear advantages, particularly at high sparsity levels. Compared to vanilla pruning, our regularized approach achieves superior generalization capability, while maintaining the same level of sparsity. Additionally, for the ImageNet dataset, we observe faster convergence and higher accuracy with the regularized method. For the CIFAR-10 experiments, our method also yields improvements in generalization compared to vanilla pruning at various sparsity levels.

Pruning Procedure: In the experimental evaluation, due to its effectiveness in promoting model convergence (see Section 2.3.4), Exponential Decay was set as the pruning schedule to compare different pruning methods/strategies on how to determine $p(t)$ percent of weights at each iteration t. *Global Pruning* which prunes the lowest $p(t)$ percent of weights [30], *Uniform Pruning* that prunes $p(t)$ percent from each layer [7], *LAMP* where weight normalization is employed for pruning $p(t)$ percent of normalized weights [8] globally, and finally, our method, *Regularized Pruning*, which uses the proposed neuron or channel score, combined with the weight normalization, modulates the outputs of channels, and prunes $p(t)$ percent of weights globally at each iteration. For GPT-2 model, we compare our method with Wanda [12]. Wanda pruning is based on l_2-norm of activations and weight magnitudes. The pruning at each step of the training is determined by the pruning function $p(t)$ according to the pruning schedule. No weight rewinding is used. We compare our method, with LAMP, Uniform, and Global pruning for CNNs and with Wanda [12] for GPT-2.

Training Procedure: We use the same hyperparameters for fairness among each pruning method. We change the learning rate from model to model to adapt to the model size. We use PyTorch [31] as training framework.

Datasets and Models: CNN architectures (pretrained with *ImageNet*) are tested using *CIFAR-10* and *ImageNet* dataset. *CIFAR-10* dataset is evaluated on models finetuned with transfer-learning: VGG-16, VGG16-BN , ResNet-18, EfficientNet-V2s, MobileNet-V2, and Inception. Models are trained for 50 epochs. Similarly, *ImageNet* dataset is finetuned using pretrained VGG-16, VGG16-BN , ResNet-50, EfficientNet-V2s, MobileNet-V2, and Inception for 10 epochs. For GPT-2 architecture, we finetune the pretrained model (with OpenAI [27] internal database) on *OpenWeb* dataset for 6000 steps.

2.3.1 Results on CIFAR-10

The test results for the CIFAR-10 dataset are provided in Figure 2.4. Given a pruning percentage, the results on the CIFAR-10 dataset show that regularized

pruning outperforms other methods most of the time. For high pruning percentages (above 99%), we observe significant difference in the final accuracy of the model. In MobileNet-V2, for instance, our method achieves 13.42% accuracy difference compared to LAMP while other methods fail. As illustrated in Figure 2.4, we observe a similar difference between our method and the baseline method in various models with high pruning percentages.

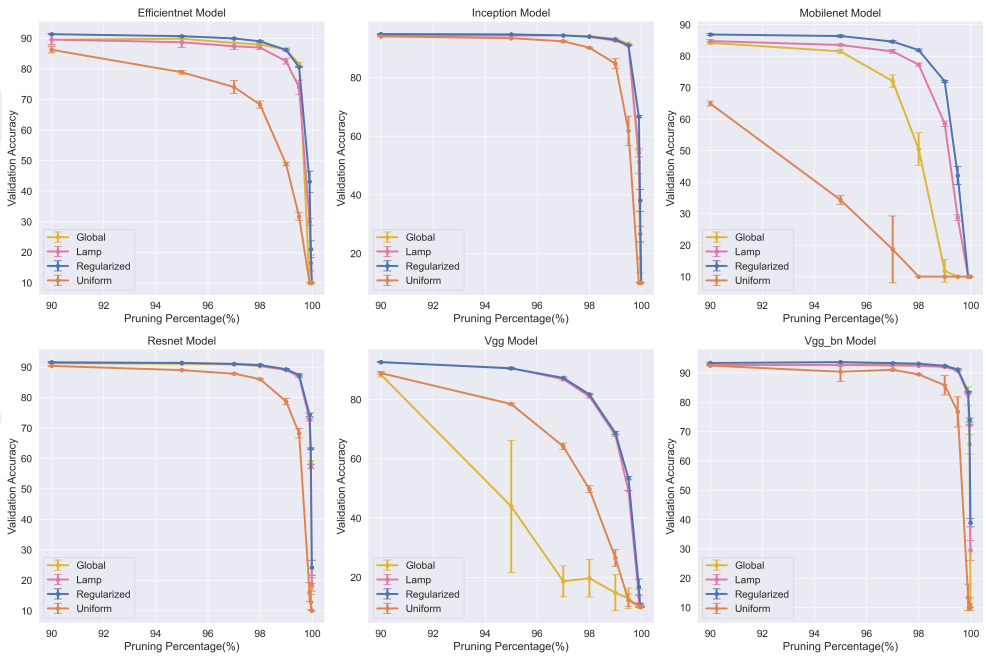


Figure 2.4: Pruning percentage vs. validation accuracy for various models trained on CIFAR-10 dataset (averaged over 5 runs)

2.3.2 Results on ImageNet Dataset

Due to limited computational resources, with ImageNet dataset, we focused our comparison only on two methods and evaluated our method against LAMP. We pruned-trained a ResNet-50, Mobilenet-V2, VGG16-BN, VGG-16, EfficientNet-V2s and Inception-V3 models for 10 epochs with pretrained models. Despite the resource constraints, our method demonstrated clear superiority in terms of accuracy and convergence speed (see Table 2.2). The validation loss plot, provided in Figure 2.5, illustrates the comparative performance over the training period. Our regularized pruning method most of the time outperforms LAMP,

Table 2.2: Validation accuracies trained on ImageNet dataset

Model	Pruning (%)	Regularized	LAMP
ResNet-50	95	**68.7**	66.7
	99	**50.0**	46.8
	99.5	**33.7**	29.1
Mobilenet-V2	60	**67.5**	67.1
	65	0.1	**62.9**
	75	0.1	**60.4**
VGG16-BN	95	**68.4**	68.1
	99	**58.3**	58.2
	99.5	**49.1**	48.6
Inception-V3	80	**72.7**	72.3
	90	69.3	**69.5**
	95	63.8	**65.2**
VGG-16	95	**67.0**	66.8
	99	50.6	**51.0**
	99.5	**41.4**	40.4
Efficientnet-V2s	80	**71.9**	68.5
	90	**63.6**	57.4
	95	**53.4**	43.4

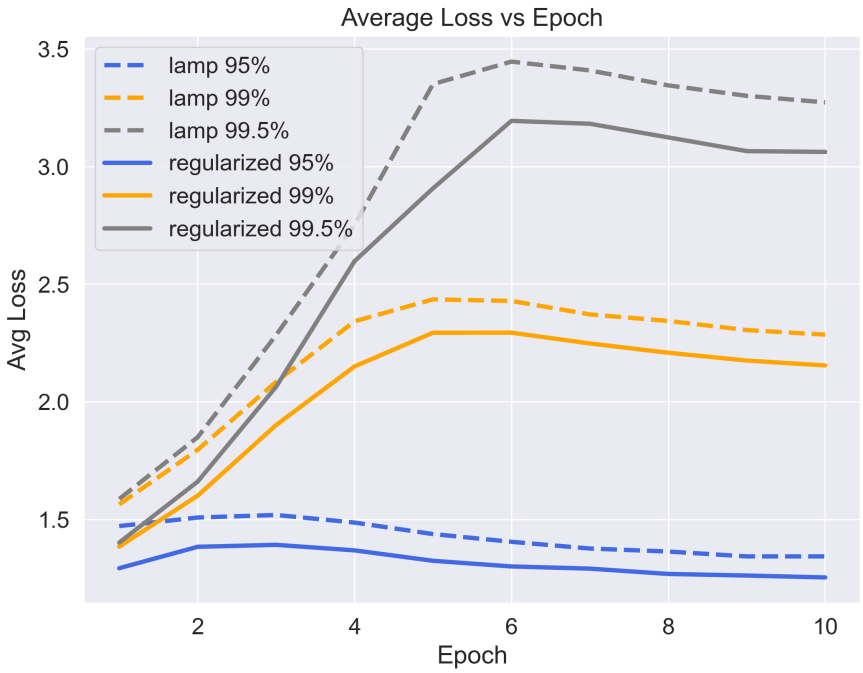
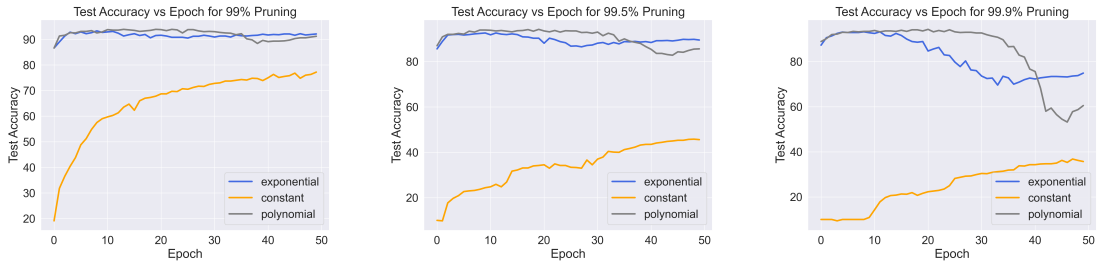


Figure 2.5: ResNet-50/Imagenet validation loss for 95%, 99% and 99.5% pruning

particularly at higher pruning rates. However, as can be seen in Table 2.2, our method fails on Mobilenet-V2 since LAMP mechanism prevents a layer being entirely pruned. On all other models, it achieves better results in majority of all tested pruning rates even with limited number of training epochs, indicating its robustness and effectiveness in preserving model performance under extreme pruning conditions. These results highlight the effectiveness of our regularized pruning strategy in enhancing model performance on large-scale datasets like ImageNet. The ability to maintain high accuracy even at very high pruning rates demonstrates the potential of our method for practical applications.

2.3.3 Results on OpenWeb Dataset

We use pretrained GPT-2 model with 124 million parameters OpenWeb dataset to compare our pruning method and Wanda pruning, we finetune and prune the models for 6000 steps. For fair comparison, similar to previous experiments, we use consistent hyperparameters in two different pruning methods. As can be seen



(a) 99% Pruning (b) 99.5% Pruning (c) 99.9% Pruning

Figure 2.6: Validation accuracy vs. epoch of VGG-16 model for different pruning schedules trained on CIFAR-10 dataset

from the Table 2.3 our method outperforms in terms on validation loss compared to Wanda pruning method on most of the cases.

Table 2.3: GPT-2 on OpenWeb dataset validation loss

Pruning (%)	Regularized	Wanda
90	**6.21**	6.75
95	6.92	**6.79**
97	**7.06**	22.78
99	**7.25**	8.99

2.3.4 Comparison of Pruning Schedules

Table 2.4 provides the comparison of the three aforementioned pruning schedules: exponential decay, gradual pruning, and constant rate pruning. The results show that exponential decay gives better results than the other two methods. As seen in Figure 2.6, the exponential schedule does not hinder the search space at the beginning of the training process. It keeps the model parameters towards the end and prevents sudden drops like gradual decay in high pruning regimes.

Table 2.4: Pruning schedule validation accuracy for VGG-16 and ResNet-18 models trained on CIFAR-10 dataset for 50 epochs

Model	Pruning (%)	Exp.	Grad.	Cons.
VGG-16	99	**92.1**	91.2	77.2
	99.5	**89.5**	85.6	45.6
	99.9	**74.9**	60.5	35.7
ResNet-18	99	**87.7**	85.2	83.1
	99.5	**84.0**	80.6	77.4
	99.9	**64.7**	52.9	54.5

2.4 Conclusion

In this chapter, we introduce a novel method for neural network pruning and regularization through a trainable neuron ranking technique. Our approach uses a weight disturbance mechanism to measure neuron importance, enabling selective pruning that maintains or improves model accuracy. We demonstrate significant performance gain across various neural network architectures and datasets, particularly at high pruning percentages. The method is easily integrated with existing training processes and can be adopted within popular deep learning frameworks. By incorporating a dynamic pruning schedule and combining structured and unstructured pruning methods, we achieve more efficient neural networks suitable for deployment on resource-constrained devices. Future work includes extending this approach to other neural network types, optimizing the pruning schedule, and addressing potential biases introduced by pruning. Our trainable neuron ranking method offers a practical solution for creating efficient neural networks without compromising their performance.

Chapter 3

Floating Point Compatible Quantization for Lightweight Neural Networks

Deep learning (DL) is taking part in every aspect of life, from medical diagnosis to speech recognition, robotics to self-driving cars, covering many industrial and commercial fields. In parallel with this spread, the amount of data to be processed is also growing while deep learning models are getting larger and more complex every day. Today, the employment of state-of-the-art models such as LLMs requires the storage of billions of parameters [32], [33]. Such large models need extensive training data, leading to high computational costs for both the training and inference stages. Most of this cost comes from vast amounts of data that need to be moved within the computation flow of these stages. Large data movement create a bottleneck for the computation, resulting in limited throughput and increased energy consumption. Since (1) DRAM access is required to compute/store large models commonly, and (2) the read/write operation of such devices is expensive in terms of both power and delay, *memory accesses constitute the majority of the inference cost* ([34]). In edge devices, such large models become very difficult or sometimes even impossible to run because of limited memory capacity. Therefore, compressing/reducing the memory requirements of

networks is crucial for (1) enabling deep learning inference for edge devices and (2) achieving efficient computation on larger platforms.

For many machine learning frameworks, an application is commonly constructed by deploying a pre-trained model using transfer learning. As these original models are usually trained on extensive datasets covering a large corpus, rather than a narrow use case, the model complexity needed to accurately represent this full input space also introduces redundancy. This leads to unnecessary computation and exacerbates aforementioned memory problem [35], [2] [36]. Real-life applications are restricted to a subset of the input space, requiring models that only require a small portion of the data [37]. Hence, they do not actually need that complexity, and there is indeed a lot of opportunity for size reduction. *Therefore, designing novel techniques that can shrink memory size by carefully eliminating unnecessary parameters of a model (while still keeping the same accuracy levels for pre-specified subtasks) is very important and essential.*

As studied extensively in the machine-learning community, model size reduction can be achieved by reducing the precision of weight representation as well. However, many state-of-the-art precision-reducing /quantization methods are not compatible with most hardware and require special architecture design [38], [39], [40]. Deep neural networks are commonly trained and run (inference stage) on 32-bit floating point representation (FP32) [13] hardware. This leads to a non-optimal hardware-software interaction when variables are quantized and coded/represented using a special numerical system like integer quantization [41] which has limited hardware support. Namely, the proposed compression-decompression format can be deployed by every hardware having a floating point unit without a specific design.

This chapter introduces a compression technique that uniquely combines pruning and quantization of neural network weights, optimizing them for efficient compression-decompression and broad hardware compatibility using the FP32 format.

Step 1: In the first phase of the algorithm, our method performs a ranking

computation that evaluates the impact of perturbations, referred to as "noise", on the weights. By systematically altering layer weights slightly, it observes the resultant error/deviation at the output values. Weights that cause significant changes at layer outputs are considered crucial and are assigned a high importance score. Conversely, weights that have minimal impact on output change are scored lower and become possible candidates for pruning. This approach not only reduces the model size by eliminating less critical weights but also prepares the model for the subsequent step that reduces the precision of the remaining weights with negligible loss accuracy. ***Step 2:*** As the second phase of our technique, the precision of the weights are altered by adjusting the mantissa bits. Based on the score assignments, starting from the least significant bits, some of the mantissa bits are set to 0. ***Step 3:*** Then, in the last stage of the compression mechanism, the remaining mantissa bits within the same filter (convolutional kernels belonging to the same output channel) are adjusted. This step reduces the variety/uniqueness of weights, inherently introducing different quantization levels, determined by the score of weights and remaining mantissa bits. Finally, the memory storage format employed for our bit adjusted model and present the compression rate formulation.

To test our pruning and quantization scheme, we selected 200 random classes from the Imagenet [25] and created a subset of the Imagenet dataset. We also tested our method on CIFAR-10 [24] dataset, applied transfer learning methods to construct different models. We used pre-trained VGG-16([18]), ResNet-50 ([19]), Inception ([23]) and Mobilenet ([22]) as base models. Results show that our compression scheme can create up to 26% accuracy difference with more compression.

The rest of the chapter is organized as follows: In Section 3.1, related studies on quantization and pruning techniques in the literature are briefly discussed. In section 3.2, we explain the details of our scoring and pruning algorithm. Proposed bit-adjustment scheme for floating point models is illustrated with formulas. We propose an efficient storage format for the proposed compression framework. Finally, in Section 3.3, simulation results for the different models are provided and

in Section 3.4, we present our conclusions and potential areas for further improvements.

3.1 Related Work

3.1.1 Pruning

A variety of pruning techniques have been developed to eliminate redundant parameters in neural network models. One of the foundational approaches by LeCun et al. [1] introduces analytical saliency, which involves approximating the Hessian of the loss with respect to the weights to estimate the impact of removing specific weights. The model is then pruned and trained iteratively until convergence. Similarly, Han et al. [5] associate importance directly with the magnitude of weights, demonstrating substantial reductions in model size for VGG-16 trained on ImageNet. Similar to LeCun et al. [1], Molchanov et al. [2] apply a saliency approximation based on the first-order Taylor expansion of the loss change when a parameter is removed. Yeom et al. [36] utilize Layer Relevance Propagation (LRP) as introduced by Bach et al. [42] to compute an importance factor for layer weights. Yu et al. [43] derive a score from the final layer's features, which is then backpropagated to assess the importance of weights throughout the model. These methods aim to identify the most effective structure for removal by estimating weight importance, yet they generally overlook the immediate impact of perturbed weights on intermediate layer outputs. Both Molchanov et al. [2] and Yeom et al. [36] focus on pruning models trained through transfer learning, whereas the other studies like those by LeCun et al. [1], Yeom et al. [36], Bach et al. [42], and Yu et al. [43] develop pruning techniques for models trained from scratch.

3.1.2 Quantization

Several quantization techniques in literature try to mitigate the memory limitations of small devices and reduce the data bit-width of deep learning models. These studies show that high precision weights are unnecessary for most models and report the advantage of using quantization techniques to accelerate and reduce deep learning models' inference and training requirements. In a paper by Han et al. [30], authors propose a weight grouping scheme for neural network quantization. In Wang et al. [44], a neural network training scheme with quantized weights is presented. As an extreme technique, Courbariaux et al. [45] presents a model based on binary neural networks. The work by Jacob et al. [41] implements 8-bit integer quantization after training the model with overhead of layer scaling.

Some of the quantization methods require special hardware. However, most processing hardware (CPUs, GPUs) supports FP32 or FP16. Therefore, reducing memory requirements of a model using quantization comes with the cost of an additional hardware unit. Therefore, in this work, we try to avoid this problem and propose a method by quantizing the heaviest part of a weight representation with a simple decoding scheme that can run on a generic FP32 or FP16 processing unit.

3.2 Method

We propose a novel technique for convolutional neural networks (CNNs) that utilizes scoring-based pruning and bit adjustment (removal of mantissa bits from an FP32 number). This framework achieves significant model size reduction with minimal accuracy loss and remains compatible with most hardware, as it retains the FP32 numerical system.

In the combined model size reduction process, we first apply a pruning algorithm based on channel (neuron) scores. Pruning is executed based on the

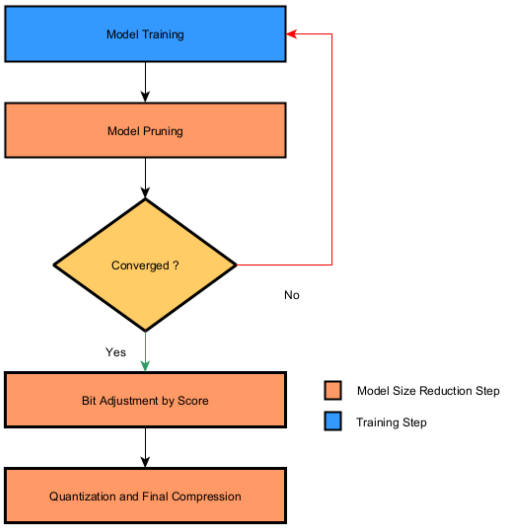


Figure 3.1: Pruning Quantization Flow

ranking of channels/neurons and the magnitude of weights. The ranking is determined by the change in the corresponding output values when a layer is disturbed. Channels that significantly affect the output when disturbed are considered more important and assigned higher scores. Once the channel scores are established (***Step 1***) and the model is pruned accordingly, we proceed with the following bit adjustment (***Step 2***) and model compression (***Step 3***) stages.

In the second step of the framework, we use the score and make bit adjustments on the mantissa of the weight. Because of the overhead in weight granularity quantization (determining quantization and bit adjustment for each weight) or bit adjustment mapping, we make a coarser bit adjustment on weight groups (i.e output channels of a convolutional layer). This step reduces the model size when applied after pruning. Additionally, it reduces the diversity among mantissa.

In the final step, the model is compressed using a sparse compression format. Since we reduce the weight diversity as well, the required storage size decreases significantly. Note that neither bit adjustment nor quantization does require any extra hardware support or a different numerical representation system. This has two benefits. First, there will not be any special hardware support requirement. Therefore, the proposed scheme is easy to implement on various devices. Second,

simple compression allows simple decompression. As a result, decompression of a compressed model can be done smoothly. The process of model compression is summarized in Figure 3.1.

Why is pruning applied as the initial step?

Pruning is performed before bit adjustment and quantization, as it reduces the model size significantly and prepares the model for subsequent steps. Pruning also ranks the channels of the model layers and decreases the number of unique weights. It is important to note that pruning relies on distorting the weights by eliminating the entire mantissa field of the FP32 weight. Bit adjustment eliminates the mantissa bits of FP32 representation and narrow their range. Consequently, applying bit adjustment prior to pruning may lead to sub-optimal results.

Pruning as the first step offers the advantage of reducing the number of unique weights. Pruning step also guides the model to train with minimal mantissa precision by penalizing significant changes when the mantissa is removed and creating a ranking among channels or neurons. After pruning, we adjust the mantissa bits of each weight based on previously calculated scores in a process we name bit adjustment. This step limits the precision of mantissa bits, thereby reducing the unique weights in the model. As previously mentioned, models typically do not require full FP32 precision. Hence, adjusting this precision without compromising accuracy enables further model size reduction. By eliminating the remaining mantissa bits of weights, we significantly advance the model size reduction.

3.2.1 Weight Pruning

For the first step, we determine weight importance using the channel score and weight magnitudes at the corresponding weight channels. We multiply weight magnitude with the channel score. Generate a pruning score for the weights and prune a certain percentage of the weights determined by the pruning schedule. Then, we train the model until it converges.

3.2.1.1 Scoring

Model compression steps starts with training and scoring-pruning process as in previous chapter. A measure of change of the channels (or neurons) is employed to rank them as in previous chapter. First, the weights of a layer are perturbed. Next, the perturbed output with the original output is compared. If the original output differs a lot by the original one, those channels get high score. Similar to previous chapter a scoring function that is differentiable (for training purposes) is adopted also giving a ranking among channels.

Similar to the calculations in Chapter 2, given an input x, weights w, and convolution (or linear) operation ($\otimes$) output $y^l(x^l; w^l)$, output ratio R_{out} and S_{out} are calculated as follows:

$$y^l(x^l; w^l) = x^l \otimes w^l \tag{3.1}$$

$$R_{out}(x^l; w^l, \delta) = \frac{\sum^N \sum^H \sum^W |\frac{y^l(x^l;w^l)}{y^l(x^l;w^l+\delta)}|}{NHW} \tag{3.2}$$

$$S_{out}(x^l; w^l, \delta) = \sigma(-R_{out} - \epsilon + 1) + \sigma(R_{out} - \epsilon - 1) \tag{3.3}$$

Then, given a model M with layer $f^l(x; w)$, θ as the nonlinear activation function (usually ReLU), the forward propagation is formulated as the following:

$$z^l(x^l; w^l) = S_{out}(x^l; w^l, \delta) y^l(x^l; w^l) \tag{3.4}$$

$$f^l(x^l; w^l) = \theta(S_{out}(x^l; w^l, \delta) y^l(x^l; w^l)) \tag{3.5}$$

The pruning is performed according to running average of score calculations at each step:

$$S^t_{pruning} = \frac{S^{t-1}_{pruning} + S^t_{out}}{2} \tag{3.6}$$

$$weight\_score = S^t_{pruning} \cdot |weight| \tag{3.7}$$

where $|weight|$ represents the absolute value of individual weights in a convolutional kernel or edges of dense layers. $S^t_{pruning}$ and $S^{t-1}_{pruning}$ are accumulated score of the layer at step t. The channel importance metric used at $weight\_score$ calculation is the weighted-accumulation of previous scores calculated during training process. This score is a measure of weight disturbance impact on the output of a layer. If a weight is not affected by the noise, it doesn't change the output. If a weight doesn't change the output, we can say that those weights or groups of weights (convolutional kernel or channels) aren't useful to deliver useful information to the next layers. Thus by adding a proportional noise or a rational degree of disturbance to the weights, we measure the importance of a weight, and use it for refining both forward propagation and pruning process.

3.2.1.2 Pruning Schedule

The exponential pruning schedule (as demonstrated in previous chapter) offers a balanced approach by initially performing aggressive pruning to reduce the number of weights early in training, effectively narrowing the search space. As training progresses, the pruning rate decreases, allowing sufficient weights to remain for fine-tuning. This method combines the benefits of both early reduction and later refinement, unlike one-shot pruning which removes many weights upfront, reducing model capacity too soon, and polynomial decay which prunes gradually, risking unnecessary parameters early on and a sudden drop in generalization capability towards the end.

Noise Modelling

For the score calculation, if weight becomes zero, the change at the output becomes untraceable (if the weight is originally nonzero). Furthermore, if the weight change sign or increase in magnitude by some order when noise is added, score calculation fails because, in such cases, the weight is replaced instead of being disturbed. Therefore, a proportional noise addition is preferred as weight disturbance. Making a noise proportional to a weight requires hefty computation steps. By removing the mantissa of a weight, we ease the computation of proportionate noise. Considering the distribution of weights, setting the mantissa bits of weight to zero results in a noise added (or subtracted) proportionally. This noise approximately has a close magnitude range with the parameter that the noise will be added. The sign of noise is determined by the sign of the weight. For example, noise becomes additive when the mantissa is removed, if a weight has a negative sign. If the weight is positive, after mantissa removal, the weight decreases. Thus, the noise becomes subtractive. Note that noise added to weight is the same exponential as weight. This ensures a proportionate change in the weight.

3.2.2 Bit Adjustment Policy

In the next step of model reduction, the number of mantissa bits of each weight has to be adjusted. This adjustment is determined by the score found at pruning-training step. We map the channel (neuron) score between 20 to 23 bits since the neural networks can work with this FP-32 precision. This ensures that the minimum score will map to 23-bits of mantissa removal and maximum score will map to 20-bits of mantissa removal. We call this process bit adjustment (see Figure 3.2) This step combined with pruning step, significantly reduces the weight diversity of models. Combination of weight removal and bit-adjustment allows compressed representation (where FP32 weights can be encoded as integers as explained in the next step).

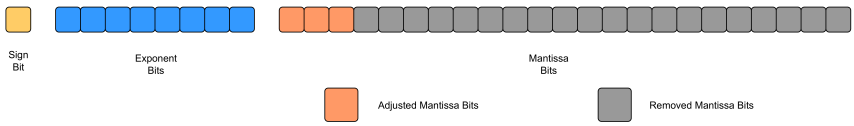


Figure 3.2: FP-32 Adjustment Diagram

With bit adjustment, we change the floating point precision of every channel. If we make the bit adjustment aggressively, the compression rate will increase. However, accuracy will drop. Thus, there is naturally a tradeoff between model accuracy and compression rate.

3.2.3 Memory Storage Format for Bit Adjusted Models

After the model is pruned based on a predetermined pruning percentage and the bit adjustment is performed on the remaining weights, weight diversity is significantly reduced, leaving only 40-200 unique weights. These remaining weights can be represented using only 6-8 bits (depending on the model and dataset) via a lookup table, where the entries correspond to the original floating-point weights. This encoding helps compress the needed storage size for model parameters. During inference time, the weights are decoded to floating point values, and model run without requiring another hardware support on a FP32 based design/architecture.

To efficiently store the pruned and bit-adjusted model, we use the Coordinate Format (COO). which stores a tensor by storing coordinate of a nonzero element and its value. Since nonzero value representation changes on our approach (we store the encoded values of FP-32), total size of the model after compression is calculated as follows:

$$S_{Compressed} = S_{Coordinate} + S_{WeightBit} \cdot N_{NonzeroWeights} \tag{3.8}$$

where $S_{Compressed}$ is compressed model size, $S_{WeightBit}$ denotes weight bit size, and $N_{NonzeroWeights}$ represents number of nonzero weights. $S_{Coordinate}$ is coordinate

representation overhead using 32 bits to represent location within sparse tensor:

$$S_{Coordinate} = 32 \cdot N_{NonzeroWeights} \tag{3.9}$$

In our method, $N_{NonzeroWeights}$ depends on the determined pruning percentage.

3.3 Experimental Results

We evaluated the effectiveness of our pruning and quantization method across different model architectures, including ResNet-50 and ResNet-18 [19], MobileNet-V2 [21] and Inception-V3 [23], using the PyTorch framework. The evaluation was conducted on two datasets: CIFAR-10 [24] and a custom subset of ImageNet-1000 [25], which we call ImageNet-200. ImageNet-200 was created as a transfer learning benchmark by selecting 200 random classes from the original ImageNet-1000 dataset and choosing 100 random images per class for the training set. This resulted in a training set of 200,000 images and a validation set of 10,000 images, drawn from the original ImageNet validation set. To demonstrate the effectiveness of our method, we applied transfer learning to all models and compared the results to static quantization method [41], which combines both magnitude pruning [5] and static quantization.

Tables 3.1 to 3.4 summarize the performance on the ImageNet-200 dataset, and Tables 3.5 to 3.12 show the overall performance on CIFAR-10 dataset.

3.3.0.1 Results on Imagenet-200 Dataset

We tested the benchmark architectures with Imagenet-200 dataset at four different pruning points with the same training configuration hyperparameters except the pruning and quantization methods. The results show that our method overperforms the static quantization on VGG model with a significant difference. However, it results in lower accuracy with a small margin with more compression

on our proposed method. Our technique achieves better results on VGG model achieving an accuracy difference up to 26% with more compression. The results for Imagenet-200 dataset is provided in Tables 3.1 to 3.4. The results highlight that even with 6 to 8-Bits of FP encoding and 3-Bit mantissa field precision, the accuracy approaches PyTorch integer quantization framework performance. This would improve as more bits are allowed in bit adjustment process, leading to more quantization levels.

Quantization Method	Model	Pruning (%)	Test Accuracy	Compression
Ours	ResNet	90	78.65	4.54
	VGG	90	78.75	4.50
	VGG_BN	90	76.35	4.51
	MobileNet	90	65.80	4.75
PyTorch	ResNet	90	84.15	4.45
	VGG	90	68.60	4.44
	VGG_BN	90	79.40	4.45
	MobileNet	90	73.90	4.51

Table 3.1: Comparison of Quantization Methods at 90% Pruning for Models Trained on Imagenet-200 Dataset

Quantization Method	Model	Pruning (%)	Test Accuracy	Compression
Ours	ResNet	95	77.40	9.08
	VGG	95	73.85	9.01
	VGG_BN	95	74.20	9.02
	MobileNet	95	55.70	9.65
PyTorch	ResNet	95	83.00	8.90
	VGG	95	56.10	8.89
	VGG_BN	95	78.40	8.89
	MobileNet	95	59.05	9.01

Table 3.2: Comparison of Quantization Methods at 95% Pruning for Models Trained on Imagenet-200 Dataset

Quantization Method	Model	Pruning (%)	Test Accuracy	Compression
Ours	ResNet	97	76.8	15.14
	VGG	97	66.85	15.23
	VGG_BN	97	70.95	15.03
	MobileNet	97	44.6	16.07
PyTorch	ResNet	97	81.0	14.83
	VGG	97	42.95	14.80
	VGG_BN	97	76.5	14.81
	MobileNet	97	0.50	15.01

Table 3.3: Comparison of Quantization Methods at 97% Pruning for Models Trained on Imagenet-200 Dataset

3.3.0.2 Results on CIFAR-10 Dataset

From Table 3.5 to 3.12 we give results for ResNet-18, VGG-16, VGG-BN16, Inception-V3 and Mobilenet-V2 models trained on CIFAR-10 dataset. The results show that for most of the cases (especially in high pruning regimes) our pruning-quantization method results in both better accuracy and compression rates. For VGG model, our method prevails where these methods struggle to maintain model integrity. Same effect is also visible on Mobilenet model. Preserving model integrity at high pruning rates make way to substantial compression rates.

Quantization Method	Model	Pruning (%)	Test Accuracy	Compression
Ours	ResNet	99	71.00	45.99
	VGG	99	44.35	45.62
	VGG_BN	99	60.90	45.67
	MobileNet	99	3.70	48.14
PyTorch	ResNet	99	73.00	44.42
	VGG	99	18.10	44.33
	VGG_BN	99	69.20	44.35
	MobileNet	99	0.50	44.95

Table 3.4: Comparison of Quantization Methods at 99% Pruning for Models Trained on Imagenet-200 Dataset

Quantization Method	Model	Pruning (%)	Test Accuracy	Compression
Ours	ResNet	90	91.43	4.52
	VGG	90	93.06	4.51
	VGG_BN	90	93.83	4.51
	MobileNet	90	86.73	4.85
	Inception	90	94.63	4.60
PyTorch	ResNet	90	92.23	4.45
	VGG	90	86.26	4.44
	VGG_BN	90	94.63	4.45
	MobileNet	90	90.00	4.51
	Inception	90	67.80	4.45

Table 3.5: Comparison of Quantization Methods at 90% Pruning for Models Trained on CIFAR-10 Dataset

Quantization Method	Model	Pruning (%)	Test Accuracy	Compression
Ours	ResNet	95	91.30	9.17
	VGG	95	91.97	9.01
	VGG_BN	95	94.07	9.02
	MobileNet	95	86.07	9.70
	Inception	95	94.67	9.19
PyTorch	ResNet	95	91.97	8.89
	VGG	95	27.07	8.88
	VGG_BN	95	94.60	8.89
	MobileNet	95	87.73	9.02
	Inception	95	63.90	8.90

Table 3.6: Comparison of Quantization Methods at 95% Pruning for Models Trained on CIFAR-10 Dataset

Quantization Method	Model	Pruning (%)	Test Accuracy	Compression
Ours	ResNet	97	91.13	15.27
	VGG	97	88.50	15.02
	VGG_BN	97	93.60	15.03
	MobileNet	97	84.87	16.16
	Inception	97	94.40	15.32
PyTorch	ResNet	97	91.43	14.81
	VGG	97	20.63	14.80
	VGG_BN	97	94.60	14.81
	MobileNet	97	10.00	15.03
	Inception	97	58.60	14.82

Table 3.7: Comparison of Quantization Methods at 97% Pruning for Models Trained on CIFAR-10 Dataset

Quantization Method	Model	Pruning (%)	Test Accuracy	Compression
Ours	ResNet	98	90.60	22.89
	VGG	98	83.60	22.51
	VGG_BN	98	93.33	22.54
	MobileNet	98	81.53	24.23
	Inception	98	94.03	22.96
PyTorch	ResNet	98	91.40	22.20
	VGG	98	10.43	22.19
	VGG_BN	98	94.40	22.20
	MobileNet	98	46.60	22.53
	Inception	98	50.30	22.21

Table 3.8: Comparison of Quantization Methods at 98% Pruning for Models Trained on CIFAR-10 Dataset

Quantization Method	Model	Pruning (%)	Test Accuracy	Compression
Ours	ResNet	99	89.00	45.70
	VGG	99	72.30	45.58
	VGG_BN	99	92.57	44.99
	MobileNet	99	71.97	48.37
	Inception	99	92.93	45.83
PyTorch	ResNet	99	89.97	44.32
	VGG	99	12.70	44.29
	VGG_BN	99	93.20	44.31
	MobileNet	99	13.13	44.97
	Inception	99	12.70	44.35

Table 3.9: Comparison of Quantization Methods at 99% Pruning for Models Trained on CIFAR-10 Dataset

Quantization Method	Model	Pruning (%)	Test Accuracy	Compression
Ours	ResNet	99.5	87.27	91.07
	VGG	99.5	56.97	90.81
	VGG_BN	99.5	91.53	90.92
	MobileNet	99.5	37.83	97.77
	Inception	99.5	91.17	91.33
PyTorch	ResNet	99.5	87.77	88.31
	VGG	99.5	10.00	88.25
	VGG_BN	99.5	91.83	88.29
	MobileNet	99.5	10.00	89.60
	Inception	99.5	10.00	88.37

Table 3.10: Comparison of Quantization Methods at 99.5% Pruning for Models Trained on CIFAR-10 Dataset

Quantization Method	Model	Pruning (%)	Test Accuracy	Compression
Ours	ResNet	99.9	73.93	448.66
	VGG	99.9	21.10	441.01
	VGG_BN	99.9	83.40	441.54
	MobileNet	99.9	10.00	474.75
	Inception	99.9	65.03	449.94
PyTorch	ResNet	99.9	67.77	428.86
	VGG	99.9	10.00	428.57
	VGG_BN	99.9	79.80	428.74
	MobileNet	99.9	10.00	435.05
	Inception	99.9	10.00	429.12

Table 3.11: Comparison of Quantization Methods at 99.9% Pruning for Models Trained on CIFAR-10 Dataset

Quantization Method	Model	Pruning (%)	Test Accuracy	Compression
Ours	ResNet	99.95	62.40	866.22
	VGG	99.95	10.37	851.38
	VGG_BN	99.95	74.30	852.40
	MobileNet	99.95	10.00	916.99
	Inception	99.95	39.30	868.68
PyTorch	ResNet	99.95	43.57	827.99
	VGG	99.95	10.00	827.37
	VGG_BN	99.95	59.87	827.70
	MobileNet	99.95	10.00	840.31
	Inception	99.95	10.00	828.43

Table 3.12: Comparison of Quantization Methods at 99.95% Pruning for Models Trained on CIFAR-10 Dataset

3.4 Conclusion

In this chapter, we present a novel approach for neural network compression that combines pruning and quantization, designed to maximize both memory efficiency and hardware compatibility. Our method is capable of reducing the size of deep learning models while making it suitable for deployment on devices with floating point based arithmetic. With weight pruning and bit-adjustment process, we observe that our approach can achieve compression rates of up to 852 times while maintaining meaningful accuracy.

The results on the CIFAR-10 and ImageNet-200 datasets, indicate that our method performs better than static quantization techniques in compression efficiency, especially at higher pruning rates. However, we observe that on more complex datasets like ImageNet-200, the generalization capacity of the pruned and quantized models slightly decreases compared to static quantization, suggesting room for future enhancements.

This section highlights the potential of combining structured pruning and quantization for model compression, while also pointing to areas where further research, such as improving generalization for complex tasks, can refine the method. Future work may focus on extending this technique to more complex architectures and further optimizing it for different hardware configurations.

Chapter 4

Arbitrary Base Logarithmic Computation for Efficient Neural Network Processing

4.1 Introduction

As the amount of data and model complexities increases exponentially, the need for efficient computing techniques becomes crucial in a broad range of artificial intelligence-driven applications. Many research points that inefficiency and redundancy of floating point based architectures makes such applications energy and time costly [34]. Therefore, in real-time applications such as autonomous driving, facial recognition, natural language processing, and robotics, the efficient processing of neural networks becomes not only necessary but crucial for ensuring fast and reliable decision-making.

Today, it has become evident that traditional floating-point representation,

while precise, often introduces unnecessary complexity in many applications. Instead of floating-point like costly systems, numerous approaches have been proposed. A prominent example is quantization of neural network [41]. With quantization, the memory and hardware requirements of artificial neural networks is aimed to be reduced. Integer quantization is a commonly used method where numbers are scaled between 0 and 1 and mapped to n-bit representation window (8-bit usually). Yet, due this representation, integer models require scaling after each layer with division or nearest bit-shift to prevent overflow in consecutive layers [41]. Furthermore, integer arithmetic requires multicycle multiplier and divider logic resulting in still high power consumption.

As an attempt to further increase the efficiency of neural networks, many authors have proposed logarithm-based approaches [46], [47], [48], [48], [49]. Pioneering this approach, [50] studies the usage of base-2 logarithmic system for multiplication and division operations. Although these studies show promising results, logarithmic computation has a fallacy when the task is addition or subtraction since addition or subtraction requires knowing the logarithm of two numbers to calculate the result. This requires bulky lookup tables, which might incur significant hardware and energy costs [49] or very narrow logarithm approximation (by using small lookup tables). In another attempt to reduce the requirement for lookup tables, [47] proposes approximations with fixed power of 2 logarithmic bases yet, this approach is still coarse and not tunable.

In this work, we propose a configurable arbitrary base, comparably accurate (to floating point representation), hardware cost efficient logarithmic computation mechanism. We present a novel logarithmic arithmetic unit that has less hardware complexity than floating point unit, with a small or no accuracy tradeoff. Our idea is based on the motivation of removing bulky division-multiplication hardware (costly in integer or floating point) and replacing the two operations with subtraction or addition of powers of the operands. The proposed calculation technique is designed to address inefficiencies, offering a more streamlined and hardware-efficient solution, which can be used from matrix multiplication in numerous fields to artificial intelligence. The implementation results in 13.97x less hardware and 9.41x less power with minimal or no decrease in model accuracy.

4.2 Method

4.2.1 Arbitrary Base Logarithmic System Arithmetic

In this section we will define the operations conducted within our arbitrary base logarithmic processing unit. Let the two operands of the floating point arithmetic be $base^a$ and $base^b$. In the proposed logarithmic system, they will be represented by the logarithm of their values (i.e. a and b) with respect to the selected fixed base in the system. (Here, we take both operands as non-negative values and keep track of the signs separately. The corresponding sign is re-added/stored after performing the arithmetic operations.)

1. Multiplication: Since the operands are now in logarithmic domain, multiplication of the two values will be calculated as follows:

$$base^m = base^a * base^b$$
$$base^m = base^{a+b}$$

2. Division: Similarly, the division of the two values will be computed as the following:

$$base^d = \frac{base^a}{base^b}$$
$$base^d = base^{a-b}$$

For further operations requiring the result of the computation in the dataflow graph, m and d will be used.

Logarithmic Approximation: Division and multiplication requires only an adder to run and provides an exact computation for the result. To enable addition and subtraction with a simpler hardware, we use linear approximations in logarithm domain with lines having slopes as powers of 2. Since the final arithmetic unit is based on logarithm domain and linear approximations, it can perform four arithmetic operations without using any divider or multiplier hardware. All arithmetic operations are computed entirely in logarithmic domain without needing any bulky lookup tables for logarithm calculations. Similar to

multiplication and division operations, the logarithm of the operands are used for the addition and subtraction operations (a and b).

3. Addition: Since all operations and numerical representations (a and b) are performed in logarithmic domain, we need to calculate an approximation (c) to represent the result in the addition/subtraction operations. Assuming $a > b$, the logarithm domain addition is written as;

$$
\begin{aligned}
base^a + base^b &= base^b(base^{a-b} + 1) \\
base^a + base^b &= base^b(base^{a-b} + 1) \\
base^c &= base^b(base^{a-b} + 1) \\
c &= b + log_{base}(base^{a-b} + 1) \\
c &= b + right\_shift(a - b, \text{shift\_by}{=}line\_slope) + line\_constant
\end{aligned}
$$

In the above expressions, the term $log_{base}(base^{a-b} + 1)$ is approximated using a linear estimations. The curve is covered by lines (defined by *line_slope* and *line_constant*) that have slopes of powers of 2 as illustrated in Figure 4.1 and effectively, eliminate and replace multiplication in the process with shift operations, owing to the line slopes with powers of 2.

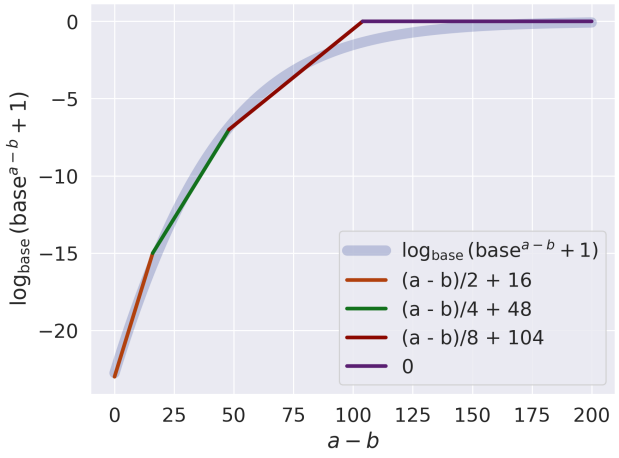


Figure 4.1: Piecewise Addition Approximation for Logarithmic Base of 0.97. Line Constants Change with the Base. *Note: We ensure that* $a - b > 0$.

4. Subtraction: Subtraction approximation requires a special care for negative

and zero results in the corresponding term $log_{base}(base^{a-b}-1)$. Thus, we first make a sign detection in hardware and switch the values in the operands (a and b) to make the logarithm term positive, $base^{a-b} > 1$ [i.e., $(a-b)<0$ when the $base<1$]. Then, similar to addition, subtraction on the logarithmic base is written as;

$$base^a - base^b = base^b(base^{a-b} - 1)$$
$$base^a - base^b = base^b(base^{a-b} - 1)$$
$$base^c = base^b(base^{a-b} - 1)$$
$$c = b + log_{base}(base^{a-b} - 1)$$
$$c = b + right\_shift(a - b, \text{shift\_by}{=}line\_slope) + line\_constant$$

The corresponding sign is stored after the computations. The approximated $log_{base}(base^{a-b} - 1)$ curve is given in Figure 4.2.

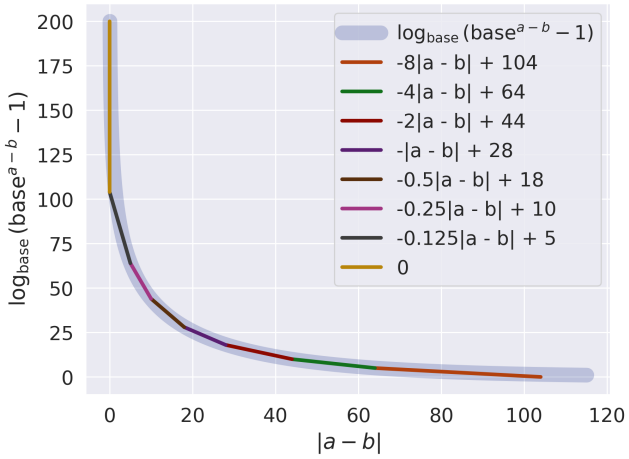


Figure 4.2: Piecewise Subtraction Approximation for Logarithmic Base of 0.97. Line Constants Change with the Base

Base selection: Another important point for our design is base selection ($base^x$). As the activation outputs of neural networks usually ranges around very small values, the base needs to be adjusted to best represent these values and is preferably chosen to be a non-negative value for simplicity (We employ a separate sign tracking process for operations as explained in the previous section.) The approximation works well when base is close to 1. As it moves away from 1,

approximation spans a wider window yet becomes more inaccurate (similar to the difference between 0.5^x and 0.9^x curves, for instance). If base becomes too close to 1, logarithmic base can not represent broad range of numbers due to reduced slope in the curve of the $base^x$ function, and requires more bit length in the hardware for x to represent larger values. Therefore, the best fit of base and minimal representation bit length have to be balanced according to the accuracy requirements. Note that the base selection is dual around 1, requiring only a sign change in the exponent. (The explanation of numerical representation format and the bit length selection process are discussed in detail in the following sections.)

Using a configurable approximation has two advantages for the method we propose. First one is, since the base is arbitrary, we can use the best fit for the arithmetic operation at hand. The implementation provides a configurable accuracy according to the workload. This increases versatility of hardware and allows optimal choice of base and logarithmic representation. The second advantage is that conversion from one logarithmic base to another one is simple. It just requires a shift to the logarithm and the boundaries of the linear approximation changes. For example; $0.97^a = 0.95^b$ where $0.97 = 0.95^s$. This implies that $0.97^a = 0.95^{b+s}$. *However, we do not cover nor need such conversions in this work.*

Finding optimal boundaries for linear approximation: As explained before, for addition and subtraction operations, we approximate the terms $log_{base}(1 + base^x)$ and $log_{base}(1 - base^x)$ correspondingly, using lines that have slopes of powers of 2 (i.e. 1, 1/2, 1/4, 1/8, 2, 4, 8). Since the slopes of the lines are pre-determined, in order to compute the linear fitting (piecewise linear approximation), we only need to find the constants of the lines and the boundaries that define the approximation regions, i.e. where each line takes active (see Figure 4.1 and 4.2). To find the optimal boundaries and line constants, we use a geometrical approach based on the idea that optimal line sets lie in between outer intersection and inner intersection. We define the outer intersection of the lines as the set of lines that are tangent to the concave up logarithmic curve and intersects outside, while inner intersection is defined as the set of lines (with the same

slope) that cuts the logarithm curve and intersects at the $log_{base}(1 \pm base^{a-b})$ from inside. Then, we start searching for optimal line constants based on an heuristic algorithm that minimizes a custom loss metric, which is defined as l_2 distance between tangent points at the curve and line. The algorithm search for the set of constants that minimize the total distance between outer and inner lines and thus the curve (see 1). This process is similar for both addition and subtraction curve approximations.

Algorithm 1 Optimization Algorithm for Line Constant and Boundary Determination

1: Set the line constants (powers of 2)
2: Find the outer bound (Find the constants of the lines tangent to $log_{base}(1 \pm base^{a-b})$ and intersecting outside of the concave up logarithm curve.)
3: Find the inner bound (Find the constants of the lines (with the same slope) that cuts $log_{base}(1 \pm base^{a-b})$ and intersects at the the logarithm curve from inside.)
4: **for** each line combination constrained by the outer and inner bounds **do**
5: Calculate the loss, the total distance between the lines and the tangent points at the curve defined by the outer bound (using a small step size for parameter sweep)
6: **end for**
7: Select the constant combination that minimizes the loss
8: Generate boundaries with the intersection of the lines (create a piecewise linear approximation)

Numerical Representation Format: In our logarithmic processing unit, an n-bit signed integer is needed to represent the exponent value (x), and one extra bit is used to indicate the sign of the number in real domain. The custom format is visualized in Figure 4.3. Addition or subtraction decision is based on the most significant sign bit. Since the power value is limited by the n-bit signed integer format, *zero* is approximated by $base^{max\_value}$ (where the *base* is a value less than but close to 1). Hence, real numbers smaller (in absolute value) than this threshold are all treated as the same. Similarly, infinity is represented as $base^{min\_value}$ where *max_value* and *min_value* are maximum and minimum numbers representable by an n-bit signed integer.

As mentioned above, the FP-32 values smaller than the threshold are regarded as *zero* within the logarithmic processor (represented by the $base^{\text{max\_value}}$). This

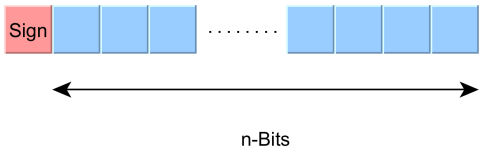


Figure 4.3: Numerical Representation Format

means that when logarithmic representation does not cover very small numerical values, the quantization accuracy of zero and near-zero values depends on the hardware configurations. However, as discussed in the following section and shown in the experimental results, this can be easily configured not to disturb the overall performance of neural networks.

Configuring Base and Representation Bit Length: To determine the suitable base for the task at hand, the worst case scenario should be considered because, with logarithmic system, the accuracy partly depends on the zero approximations and this is directly adjusted/approximated with the n-bit *max_value* representation in the custom numerical format. Therefore, we investigate the minimum nonzero output within the neural network workload (layer activation outputs) and choose *base* and bit length (n) pair accordingly. The motivation behind this idea is to tune inherent quantization mechanism within the logarithmic hardware and prevent model deviate from its original prediction capability.

If *base* is chosen too close to 1 and the maximum representable power value is limited (i.e., number of bits, n is constrained by hardware specifications), the *zero* will be set to a comparably large number (due to the slow change in the curve of the $base^x$ function and limitation in representable maximum x value). For example, with a base 0.99 and 8-bits of representation, *zero* (or minimum number within logarithmic system) becomes $0.99^{127} = 0.208$ which is a fairly large number to represent zero for neural networks.

In the hardware, numbers smaller than $base^{max\_value}$ (threshold) are considered zero and operations are handled accordingly. For example, multiplication with zero yields zero in logarithmic domain. A direct effect of this scenario is the inherent ***pruning*** within the hardware. Since numbers smaller than the threshold

are considered zero, hardware treats such weights or inputs as zero. Another effect of logarithmic arithmetic is inherent ***quantization*** effect. If the representation is coarse (bit length is not enough), then the neural network might see some accuracy drop as illustrated in simulation results.

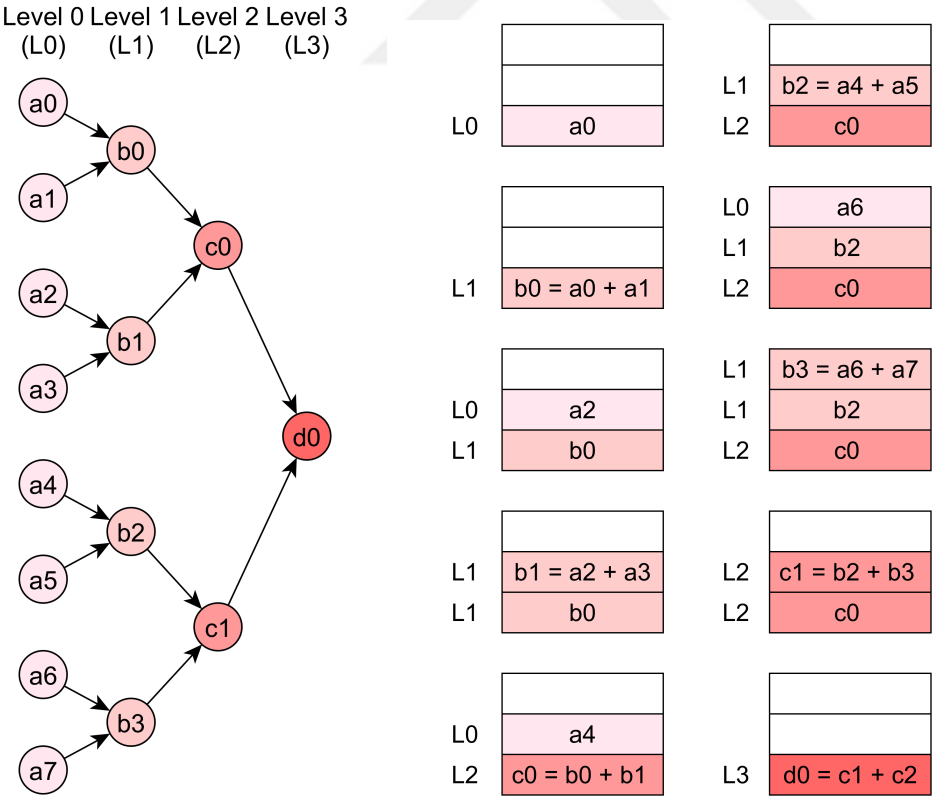
4.2.2 Hardware Implementation

In this section hardware design of the arbitrary base logarithmic PE will be described.

1. Sequential Tree Adder: Addition and subtraction approximations do not work well if two operands have different orders of magnitude. If one of the operands has a different order of magnitude than the other, smaller one gets neglected and result becomes the one with the large order of magnitude compared to the other operand. Namely, if one of the operands has larger absolute value, other operand will be cancelled (see the lines with near zero slopes in Figure 4.1 and 4.2). Yet, in a MAC (multiply and accumulate) architecture, two inputs come into the unit, get multiplied and the result is added to previously accumulated value. If the accumulated value is larger than the result to be summed with, linear approximation to the $log_{base}(1 \pm base^{a-b})$ becomes deviated from the original result resulting in some accuracy loss. Thus, to reduce/eliminate the probability of such cases, a hardware design that sums sequential inputs in a tree manner is developed. This accumulates numbers (adds/subtracts) in a balanced manner according to the logarithmic addition/subtraction approximations. If a register contains result of tree depth level i, we can say that it has most probably same order of magnitude with another register containing result of level i. Therefore, we can sum two registers with results of level i and combine the new result as $i + 1$. Therefore, results become more accurate for dot product.

In a sequence with length 2^l (number of inputs to be summed), there are l levels, where level numbers also indicates the number of summed inputs at the registers (see again Figure 4.4a). For example, if l is equal to 3, tree has 2^3

leafs and the register at the level 2 stores the sum of 8 inputs (see Figure 4.4a). We continue this process until the final level, l, is reached and all inputs are summed/reduced. To balance the order of magnitude, in a traditional tree sum architecture, 2^l storage is required (l is the depth of the tree to be summed, see Figure 4.4a). However, with the proposed new design, l registers are sufficient to accomplish the tree sum. The nodes of the tree reduces themselves. Sequential tree adder works by storing the levels of tree in addition to the intermediate/input values to be summed. As soon as they become available/computed, data at same levels are summed and collapsed into one register in the array (old data register is overwritten) (see Figure 4.4b). New intermediate data (MAC output) that is at a smaller level than the existing data in the array is stored into an empty register pointed by the control logic. The reduction mechanism is illustrated in Figure 4.4b and Algorithm 2 in detail.



(a) Traditional Tree Sum (b) Sequential Tree Sum

Figure 4.4: Sequential Tree Sum Algorithm Example for $2^3 = 8$ Inputs

To ensure those rules works correctly, there is a pointer to the critical indexes

Algorithm 2 Sequential Tree Sum Algorithm

Given n registers
Store the levels of those registers
while two registers have the same level **do**
    Sum the registers with the same level into the smaller indexed register
    Increase the level of the smaller indexed register
    Assign the higher indexed register to take the input
end while
Use a pointer to denote the storage location of the input
Rules:
if the level is 1 (1 input is stored in the register) **then**
    Add new input to this register data
    Make the level 2
else if there is no reduction (two same leveled registers are not reduced) **then**
    Store the new data into the lowest indexed empty place
else
    Store the new data into the higher indexed register
    Load the result of the reduction to the lower indexed register
    Make the level of the lower indexed register $l + 1$
    Make the level of the higher indexed register 1
end if

in the register file. This pointer is controlled by a state machine that realizes Algorithm 2.

2. Systolic Array Architecture: To test the effectiveness of our method, we implemented a systolic array where each processing element (PE) contains a ***logarithmic processing unit*** that is able to perform the four basic arithmetic operations in logarithmic domain. The array dataflow is designed to be output stationary where each PE is responsible from creating an output element by accumulating the output data with the explained ***sequential tree adder*** structure.

3. Logarithmic Processing Unit: In this section, the logarithmic processing unit in each PE that achieves an approximate multiply and accumulate operation will be examined. For this purpose, assume that there will be two inputs ($base^x$ and $base^y$) to be multiplied (assume that the results is $base^a$) and accumulated with the value residing in the storage element ($base^b$) of the logarithmic processing unit. By using the approximation, we find c representing the accumulated value for $base^c = base^b(base^{a-b} \pm 1)$. Overall design is visualized in Figure 4.5 where bold lines demonstrates dataflow within the unit and dashed lines represents control signals.

Zero Handling and Overflow Check: Two inputs in real domain (input $base^x$ and input $base^y$) are multiplied in real domain by adding their powers in logarithmic domain. By doing so, $base^a$ value is obtained. However, there are two cases to consider when finding a. First case is, if $x + y$ is less then the representable minimum value, there becomes an overflow. For this case, a is set to the minimum integer within the representable range. Another case is when x or y is zero. If one of x or y is minimum integer within the representable range, this case is considered as multiplication with zero and set a to zero in real domain.

Control Mechanism: We realize the Algorithm 2 using pointer and level registers on hardware as visualized in Figure 4.4. This unit controls the dataflow from data register, input and sum unit. The key idea is to choose the input of sum unit and data register.

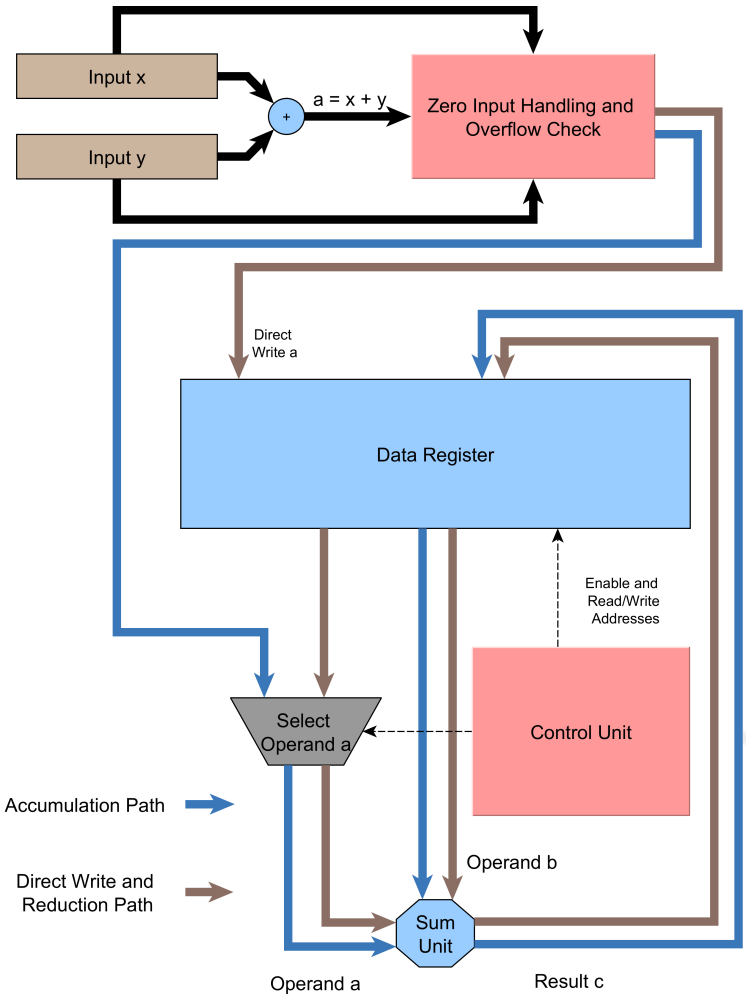


Figure 4.5: Logarithmic Processing Unit

Sum Unit: In this unit, addition or subtraction approximation is done. This unit takes two inputs, a and b, and generates an approximation c for logarithmic addition or subtraction ($base^c = base^b(base^{a-b} \pm 1)$). As mentioned before, adders and shift operation are used to compute the approximation value c, where the piecewise lines in Figures 4.1 and 4.2 are realized with bit-shift (for slope) and addition (for line constants) operation (see Figure 4.5).

Datapath: The dataflow is visualized with bold lines and datapath elements are marked with light-blue color in Figure 4.5. *Data Register* stores the values for sequential tree sum algorithm (see the array structure in Figure 4.4b) and *Sum*

Unit realizes addition or subtraction approximations for the accumulation or reduction steps (see Figure 2). Storing the result of *Sum Unit*, c, and the multiplied result, a, (corresponding to addition in logarithmic domain) to *Data Register* is orchestrated by *Control Unit*. The multiplexer *Select Operand a* chooses whether the *Operand a* of *Sum Unit* is from *Data Register* or multiplied data (to choose between accumulation or reduction step).

4.3 Test Methodology

For the simulations of our approach, we used MNIST [51], CIFAR-10 [24] and Imagenet-1000 [52] datasets. First, an RTL design of the architecture was created as described in previous section. To avoid the slow hardware HDL simulation (much slower compared to actual hardware processing), a faster hardware emulation of the logarithmic processing was implemented in software to collect accuracy results for various benchmarks easily. For this purpose, we have created dense and convolutional layers in CUDA. For power and area estimation (utilization - LUT - FF), the Verilog source of the 8x8 systolic array architecture was used with logarithmic PEs and a Floating Point Unit [53] design.

In next section, the results of experiments on MNIST-Net, VGG-16 [18] and ResNet-18 [19] models with MNIST [51], CIFAR-10 [24] and Imagenet-1000 [52] validation datasets are provided. Power and hardware utilization estimations are based on 8x8 systolic array architecture with interconnect buffers. The estimations are generated on Vivado 2021.1 using AMD Zynq Ultrascale+ MPSoC EV XCZU7 FPGA.

4.4 Experimental Results

Hardware costs are given in terms of lookup table (LUT) and flip flop (FF) usage for floating point and logarithmic representation with different bit lenght configuration in Table 4.1. In Table 4.2, power estimation for each hardware configuration is given. Finally in Table 4.3, accuracy results of the CUDA simulation is provided. As can be seen from the Table 4.1, Our design reduces LUT requirements 13.97 times in 10-bit configuration, 12.78 times in 11-bit configuration and 12.03 times in 12-bit configuration. Considering FF usage, our design has slightly more usage than FP-32 hardware. In terms of power, logarithmic hardware is more efficient than FP-32 hardware reducing power requirements up to 9.41 times. Comparing FP-32 and logarithmic unit in terms of validation accuracy, we observe that our method achieves same accuracy or near FP-32 accuracy where worst case is 8% accuracy drop in ResNet-18 model with 10-bit logarithmic hardware configuration, yet, even near FP-32 accuracy in 12-bit configuration for all models, which increases in parallel with the number of bits (at the cost of hardware power). As a side note, when a base is chosen very close to 1 and bitlength is increased, logarithmic system may represent more finer ranges than FP-32 range which is not adjustable in FP-32.

Table 4.1: Flip Flop (FF) - Lookup Table (LUT) Usage for Different Hardware Configurations

Configuration	**Hardware Cost**	
Bit Length	**Flip Flop**	**LUT**
FP-32	224	7353
10-Bits	247	526
11-Bits	263	575
12-Bits	279	611

Table 4.2: Power Estimation for 8x8 Systolic Array Architecture

Configuration	**Power Cost (W)**
FP-32	20.48
10-Bits	2.176
11-Bits	2.432
12-Bits	2.688

Table 4.3: Validation Accuracy Comparison For Models and Datasets

		Hardware Configuration		
Model-Dataset	FP-32	**10-Bits** *Base 0.95*	**11-Bits** *Base 0.97*	**12-Bits** *Base 0.99*
MNIST-Net	98.94%	98.95%	98.88%	98.94%
VGG-16, CIFAR-10	91.06%	89.41%	90.56%	91.06%
ResNet-18, CIFAR-10	80.86%	72.54%	79.13%	80.23%
VGG-16, Imagenet	65.70%	64.45%	64.45%	65.48%
ResNet-50, Imagenet	70.56%	63.01%	67.93%	69.89%

4.5 Conclusion

In conclusion, this work proposes a logarithmic arithmetic unit that is a possible alternative for neural networks. Using a configurable base and piecewise linear approximations to addition and subtraction, we offer both lightweight and versatile solution for logarithmic arithmetic. This allows us to achieve adjust precision for the needs of the workload. We test our idea by hardware validated CUDA based simulations on different datasets and neural network accelerators. Our findings show that logarithmic arithmetic unit is a strong alternative to floating point based architectures with much smaller hardware cost without zero or near zero compromise on model accuracy. For future works, although we tested the logarithmic arithmetic unit configurations on model granularity, our work can be tested with layer base-bit length configuration for achieving better precision and less memory. Furthermore, this hardware might be tested in general matrix multiplication applications.

Chapter 5

Conclusion

This thesis introduces three complementary methods for optimizing neural networks by hardware and software co-design, focusing on pruning, quantization, and a logarithmic arithmetic unit to improve model efficiency.

The first technique, the trainable neuron ranking approach, utilizes a weight disturbance mechanism to assess neuron importance, allowing for regularized pruning that improves model accuracy compared to other compression/pruning methods. This idea provides significant performance gains across various neural network architectures and datasets, especially at high pruning levels. By incorporating a dynamic pruning schedule and combining structured and unstructured pruning techniques, we achieve sparse neural networks that are well-suited for deployment on memory constrained applications.

In the second method, we combine pruning and quantization to maximize both memory efficiency and hardware compatibility with floating point processing units. Our approach effectively reduces the size of deep learning models, achieving compression rates of up to 852 times by preserving meaningful accuracy. Performance evaluations on the CIFAR-10 and ImageNet-200 datasets shows that our method outperforms static quantization techniques, particularly at higher pruning rates. However, on more complex datasets, we observe a slight

reduction in generalization capacity, signaling areas for potential improvement.

Finally, in the last method, a novel logarithmic arithmetic unit as an alternative to traditional floating-point architectures is introduced. By employing a configurable base mechanism and piece-wise linear approximations for addition and subtraction operations, this technique presents an adjustable and lightweight solution, achieving very close or same accuracy compared to floating-point units but with significantly reduced hardware costs. The results suggest that the logarithmic arithmetic unit is a strong candidate for neural network accelerators, particularly in resource-limited environments.

Future research can be based on exploring and extending the trainable neuron ranking method to a broader range of neural network architectures like Large Language Models (LLMs). An extended work on LLMs might help reduce datacenter costs for numerous applications on this area. Further study on the combined pruning and quantization technique can focus on increasing generalization capabilities of the framework for more complex tasks and improving the method to support more complex architectures. For the logarithmic arithmetic unit, future work may test layer-specific bit-length configurations to fine tune precision levels and minimize hardware-memory usage. Integration of logarithmic arithmetic unit in general matrix multiplication and other arithmetic-intensive tasks could further prove its efficiency and broaden its use in practical hardware implementations. Future directions jointly aim to develop and expand on the concepts presented in this thesis, pushing the boundaries of neural network optimization for both software and hardware efficiency.

Bibliography

[1] Y. LeCun, J. Denker, and S. Solla, "Optimal brain damage," Advances in neural information processing systems, vol. 2, 1989.

[2] P. Molchanov, S. Tyree, T. Karras, T. Aila, and J. Kautz, "Pruning convolutional neural networks for resource efficient inference," arXiv preprint arXiv:1611.06440, 2016.

[3] P. Molchanov, A. Mallya, S. Tyree, I. Frosio, and J. Kautz, "Importance estimation for neural network pruning," in Proceedings of the IEEE/CVF conference on computer vision and pattern recognition, 2019, pp. 11 264–11 272.

[4] N. Lee, T. Ajanthan, and P. H. Torr, "Snip: Single-shot network pruning based on connection sensitivity," arXiv preprint arXiv:1810.02340, 2018.

[5] S. Han, J. Pool, J. Tran, and W. Dally, "Learning both weights and connections for efficient neural network," Advances in neural information processing systems, vol. 28, 2015.

[6] U. Evci, T. Gale, J. Menick, P. S. Castro, and E. Elsen, "Rigging the lottery: Making all tickets winners," in International conference on machine learning. PMLR, 2020, pp. 2943–2952.

[7] M. Zhu and S. Gupta, "To prune, or not to prune: exploring the efficacy of pruning for model compression," arXiv preprint arXiv:1710.01878, 2017.

[8] J. Lee, S. Park, S. Mo, S. Ahn, and J. Shin, "Layer-adaptive sparsity for the magnitude-based pruning," arXiv preprint arXiv:2010.07611, 2020.

[9] Z. Liu, J. Li, Z. Shen, G. Huang, S. Yan, and C. Zhang, "Learning efficient convolutional networks through network slimming," in Proceedings of the IEEE international conference on computer vision, 2017, pp. 2736–2744.

[10] T. Zhuang, Z. Zhang, Y. Huang, X. Zeng, K. Shuang, and X. Li, "Neuron-level structured pruning using polarization regularizer," Advances in neural information processing systems, vol. 33, pp. 9865–9877, 2020.

[11] Z. You, K. Yan, J. Ye, M. Ma, and P. Wang, "Gate decorator: Global filter pruning method for accelerating deep convolutional neural networks," Advances in neural information processing systems, vol. 32, 2019.

[12] M. Sun, Z. Liu, A. Bair, and J. Z. Kolter, "A simple and effective pruning approach for large language models," arXiv preprint arXiv:2306.11695, 2023.

[13] IEEE, "Ieee standard for floating-point arithmetic," IEEE Computer Society, New York, NY, USA, Jul. 2019, revised 2019.

[14] C. E. Shannon, "A mathematical theory of communication," The Bell system technical journal, vol. 27, no. 3, pp. 379–423, 1948.

[15] I. Goodfellow, "Deep learning," 2016.

[16] S. Ioffe, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," arXiv preprint arXiv:1502.03167, 2015.

[17] Y. He and L. Xiao, "Structured pruning for deep convolutional neural networks: A survey," IEEE transactions on pattern analysis and machine intelligence, 2023.

[18] K. Simonyan, "Very deep convolutional networks for large-scale image recognition," arXiv preprint arXiv:1409.1556, 2014.

[19] 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.

[20] M. Tan and Q. Le, "Efficientnet: Rethinking model scaling for convolutional neural networks," in International conference on machine learning. PMLR, 2019, pp. 6105–6114.

[21] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "Mobilenetv2: Inverted residuals and linear bottlenecks," in Proceedings of the IEEE conference on computer vision and pattern recognition, 2018, pp. 4510–4520.

[22] A. G. Howard, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," arXiv preprint arXiv:1704.04861, 2017.

[23] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," in Proceedings of the IEEE conference on computer vision and pattern recognition, 2016, pp. 2818–2826.

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

[25] 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, pp. 211–252, 2015.

[26] A. Vaswani, "Attention is all you need," Advances in Neural Information Processing Systems, 2017.

[27] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever et al., "Language models are unsupervised multitask learners," OpenAI blog, vol. 1, no. 8, p. 9, 2019.

[28] A. Karpathy, "nanogpt: The simplest, fastest repository for training/finetuning medium-sized gpts," URL https://github.com/karpathy/nanoGPT, 2023.

[29] L. Gao, S. Biderman, S. Black, L. Golding, T. Hoppe, C. Foster, J. Phang, H. He, A. Thite, N. Nabeshima et al., "The pile: An 800gb dataset of diverse text for language modeling," arXiv preprint arXiv:2101.00027, 2020.

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

[31] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga et al., "Pytorch: An imperative style, high-performance deep learning library," Advances in neural information processing systems, vol. 32, 2019.

[32] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar et al., "Llama: Open and efficient foundation language models," arXiv preprint arXiv:2302.13971, 2023.

[33] T. B. Brown, "Language models are few-shot learners," arXiv preprint arXiv:2005.14165, 2020.

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

[35] F. Dalvi, H. Sajjad, N. Durrani, and Y. Belinkov, "Exploiting redundancy in pre-trained language models for efficient transfer learning," arXiv preprint arXiv:2004.04010, 2020.

[36] S.-K. Yeom, P. Seegerer, S. Lapuschkin, A. Binder, S. Wiedemann, K.-R. Müller, and W. Samek, "Pruning by explaining: A novel criterion for deep neural network pruning," Pattern Recognition, vol. 115, p. 107899, 2021.

[37] S. J. Pan and Q. Yang, "A survey on transfer learning," IEEE Transactions on knowledge and data engineering, vol. 22, no. 10, pp. 1345–1359, 2009.

[38] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "Eie: Efficient inference engine on compressed deep neural network," ACM SIGARCH Computer Architecture News, vol. 44, no. 3, pp. 243–254, 2016.

[39] J. Lee, C. Kim, S. Kang, D. Shin, S. Kim, and H.-J. Yoo, "Unpu: An energy-efficient deep neural network accelerator with fully variable weight

bit precision," IEEE Journal of Solid-State Circuits, vol. 54, no. 1, pp. 173–185, 2018.

[40] S. Jain, S. K. Gupta, and A. Raghunathan, "Tim-dnn: Ternary in-memory accelerator for deep neural networks," IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 28, no. 7, pp. 1567–1577, 2020.

[41] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko, "Quantization and training of neural networks for efficient integer-arithmetic-only inference," in Proceedings of the IEEE conference on computer vision and pattern recognition, 2018, pp. 2704–2713.

[42] S. Bach, A. Binder, G. Montavon, F. Klauschen, K.-R. Müller, and W. Samek, "On pixel-wise explanations for non-linear classifier decisions by layer-wise relevance propagation," PloS one, vol. 10, no. 7, p. e0130140, 2015.

[43] R. Yu, A. Li, C.-F. Chen, J.-H. Lai, V. I. Morariu, X. Han, M. Gao, C.-Y. Lin, and L. S. Davis, "Nisp: Pruning networks using neuron importance score propagation," in Proceedings of the IEEE conference on computer vision and pattern recognition, 2018, pp. 9194–9203.

[44] K. Wang, Z. Liu, Y. Lin, J. Lin, and S. Han, "Haq: Hardware-aware automated quantization with mixed precision," in Proceedings of the IEEE/CVF conference on computer vision and pattern recognition, 2019, pp. 8612–8620.

[45] M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv, and Y. Bengio, "Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1," arXiv preprint arXiv:1602.02830, 2016.

[46] M. Christ, F. De Dinechin, and F. Pétrot, "Low-precision logarithmic arithmetic for neural network accelerators," in 2022 IEEE 33rd International Conference on Application-specific Systems, Architectures and Processors (ASAP). IEEE, 2022, pp. 72–79.

[47] M. S. Ansari, B. F. Cockburn, and J. Han, "An improved logarithmic multiplier for energy-efficient neural computing," IEEE Transactions on Computers, vol. 70, no. 4, pp. 614–625, 2020.

[48] A. Sanyal, P. A. Beerel, and K. M. Chugg, "Neural network training with approximate logarithmic computations," in ICASSP 2020-2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP). IEEE, 2020, pp. 3122–3126.

[49] S. Vogel, M. Liang, A. Guntoro, W. Stechele, and G. Ascheid, "Efficient hardware acceleration of cnns using logarithmic data representation with arbitrary log-base," in 2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD). IEEE, 2018, pp. 1–8.

[50] J. N. Mitchell, "Computer multiplication and division using binary logarithms," IRE Transactions on Electronic Computers, no. 4, pp. 512–517, 1962.

[51] 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.

[52] 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.

[53] D. Shanley, "FPU," https://github.com/danshanley/FPU, 2024.

[54] D. Bertsekas and J. N. Tsitsiklis, Introduction to probability. Athena Scientific, 2008, vol. 1.

[55] I. Loshchilov and F. Hutter, "Sgdr: Stochastic gradient descent with warm restarts," arXiv preprint arXiv:1608.03983, 2016.

[56] G. TensorFlow, "Large-scale machine learning on heterogeneous systems," Google Research, vol. 10, p. s15326985ep4001, 2015.

[57] N. Corporation, "Nvidia cuda compute unified device architecture: Programming guide," NVIDIA Whitepaper, 2007. [Online]. Available: https://developer.nvidia.com/cuda-toolkit

[58] Yükseköğretim Kurulu Başkanlığı, "Yapay zeka kullanımına dair etik rehber," 2024. [Online]. Available: https://www.yok.gov.tr/Documents/2024/yapay-zeka-kullanimina-dair-etik-rehber.pdf

[59] Bard, "Bard: A large language model by google ai," 2024. [Online]. Available: https://ai.googleblog.com/2022/01/lamda-language-model-for-dialogue-and.html

[60] OpenAI, "Chatgpt: A large language model," 2024, https://www.openai.com/chatgpt.

Appendix A

Regularized Pruning

A.1 Discussion on Computational Overhead

There is a computational overhead coming from the score calculation algorithm due to the forward operation being repeated for disturbed output calculation. However, this overhead is greatly reduced as the model and dataset grows large as in the case of training Imagenet dataset (see Table A.2). If the model is small, the overhead incurs noticeable difference (in terms of percentage increase) in computational time compared to the version without scoring as in case of models trained with CIFAR-10 dataset (see Table A.1). Nonetheless, it is important to note that the method achieves faster convergence as provided in Section 2.3 (with less training epochs). This actually helps reduce the overall overhead if model training is continued until convergence. Considering total experiments before finalizing the method. We spend 500-700 single GPU time.

Table A.1: Average runtime (seconds) per epoch for different models and methods trained on CIFAR-10 dataset

Model	**Global**	**Lamp**	**Regularized**	**Uniform**
EfficientNet	16	17	21	16
Inception	26	27	33	26
Mobilenet-V2	6.9	7.2	8.7	7.1
Resnet-18	5.9	5.9	6.1	5.9
VGG-16	9.9	10	14	10
VGG16-BN	12	12	16	12

Table A.2: Average runtime (seconds) per epoch for different models and methods trained on Imagenet dataset

Model	**Lamp**	**Regularized**
EfficientNet	2748	3389
Inception	5693	6826
Mobilenet-V2	2141	2136
Resnet-50	2161	2492
VGG-16	3035	4429
VGG16-BN	3455	4717

Table A.3: Average runtime (seconds) per step for different methods and GPT-2 model trained on OpenWeb dataset

Model	**Wanda**	**Regularized**
GPT-2	3.08	3.11

A.2 Training Specifications

We used PyTorch Paszke et al. [31] framework for all of the experiment with the given set of hyperparameters below.

A.2.1 Pruning Tests with Imagenet Dataset

A.2.1.1 Hyperparameters

- Batch Size: 512
- Total Training Steps: 10 * Steps per epoch
- Total Training Epochs: 10
- Optimizer: Stochastic Gradient Descent
- Learning Rate: 0.01
- Training-Validation Split: 80-20 %
- τ (For Exponential Decay Pruning): 10

A.2.1.2 Hardware Specifications

- GPU: NVIDIA A100 Tensor Core GPU
- CPU: AMD 7742 2.24 GHz

A.2.2 Pruning Tests with Cifar-10 Dataset

A.2.2.1 Hyperparameters

- Batch Size: 2048

- Total Training Steps: 50 * Steps per epoch
- Total Training Epochs: 50
- Initial Learning Rate: 0.0001 for VGG-16, 0.001 for other models)
- Learning Rate Decay: 0.9 for every 10 epochs
- τ (For Exponential Decay Pruning): 10

A.2.2.2 Hardware Specifications

- GPU: NVIDIA A100-80GB Tensor Core GPU
- CPU: AMD 7742 2.24 GHz
- RAM: 1TB
- Operating System: Red Hat Enterprise Linux 8.5 (Ootpa)

A.2.3 Pruning Schedule Tests with Cifar-10 Dataset

A.2.3.1 Hyperparameters

- Batch Size: 128
- Total Training Steps: 50 * Steps per epoch
- Total Training Epochs: 50
- Initial Learning Rate: 0.0001 for VGG-16, 0.001 for other models)
- Learning Rate Decay: 0.9 for every 10 epochs
- τ (For Exponential Decay Pruning): 10

A.2.3.2 Hardware Specifications

- GPU: NVIDIA GeForce RTX 3070
- CPU: Intel(R) Core(TM) i9-10920X CPU @ 3.50GHz
- RAM: 64GB
- Operating System: CentOS Linux 7 (Core)

A.2.4 Pruning Tests with OpenWeb Dataset

A.2.4.1 Hyperparameters

- Batch Size: 36
- Total Training Steps: 6000 and 12000
- Block Size: 1024
- Gradient Accumulation Steps: 5
- Initial Learning Rate: 0.001
- Learning Rate Decay: Cosine Annealing with coefficient power of $\frac{\tau}{10}$
- τ (Used for Exponential Decay Pruning): 10

A.2.4.2 Hardware Specifications

- GPU: 2 x NVIDIA A100-80GB Tensor Core GPU
- CPU: AMD 7742 2.24 GHz
- RAM: 1TB
- Operating System: Red Hat Enterprise Linux 8.5 (Ootpa)

- Note: Other hyperparameters are same with the code given by Karpathy [28].
- Model Parameter Count: GPT-2 with 124M weights

A.3 Extra Results

In this section we provide additional results of the experiments conducted in "Experimental Results" section.

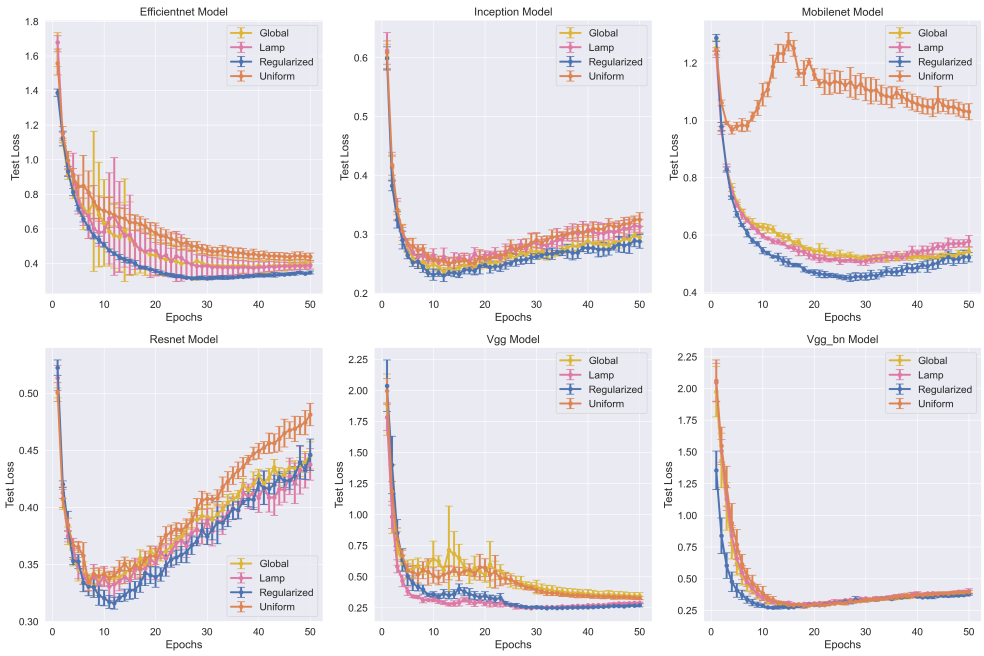


Figure A.1: Test loss vs. epoch for various models (averaged over 5 runs) trained on CIFAR-10 dataset with 90% Pruning

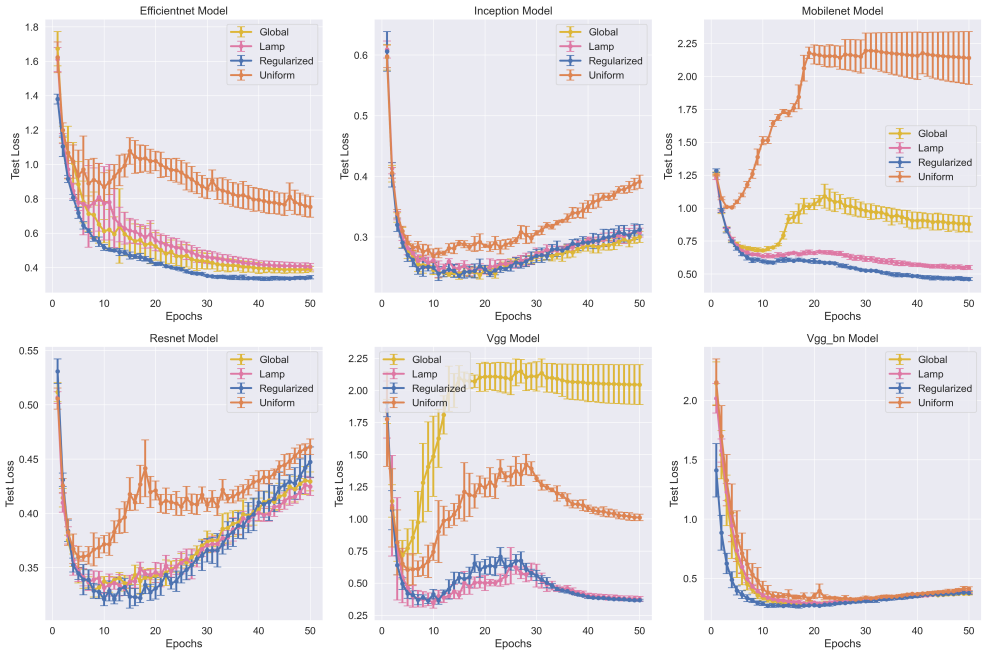


Figure A.2: Test loss vs. epoch for various models (averaged over 5 runs) trained on CIFAR-10 dataset with 97% Pruning

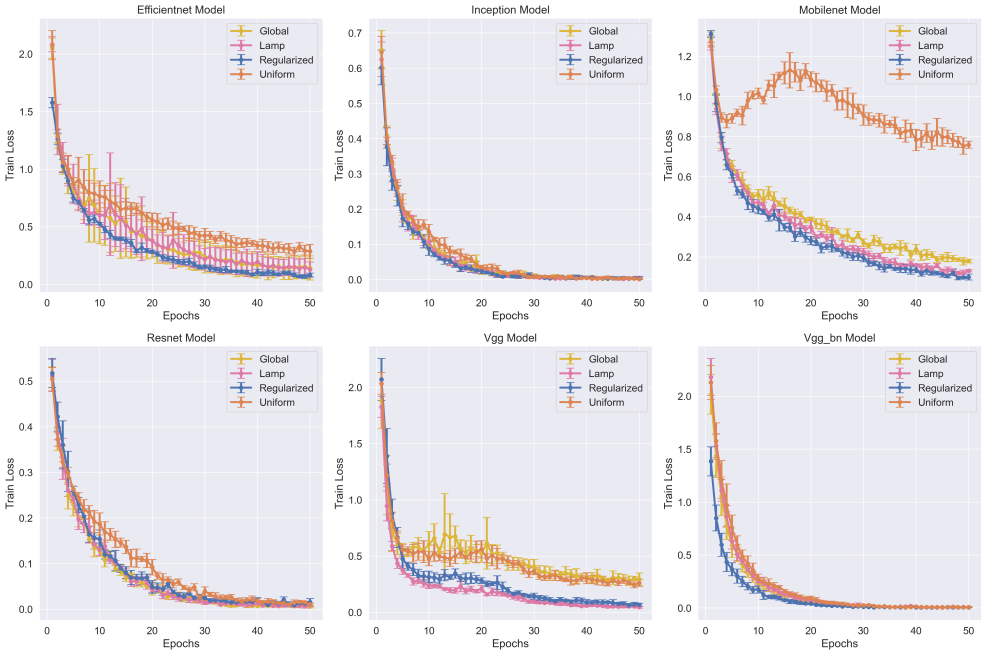


Figure A.3: Train loss vs. epoch for various models (averaged over 5 runs) trained on CIFAR-10 dataset with 90% Pruning

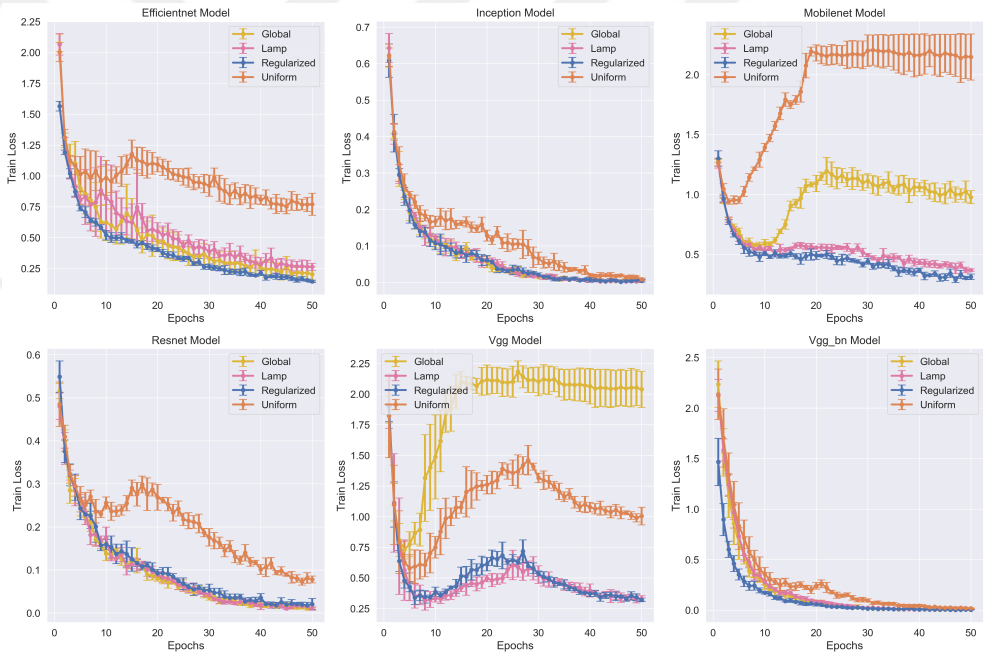


Figure A.4: Train loss vs. epoch for various models (averaged over 5 runs) trained on CIFAR-10 dataset with 97% pruning

Table A.4: Model performance on CIFAR-10 dataset for various models

Pruning Percentage (%)		90	95	97	98	99	99.5	99.9	99.95	99.99
Model	Method	Test Accuracy (%)								
EfficientNet-V2s	Global	89.53 ± 1.76	89.86 ± 0.17	88.47 ± 1.12	87.97 ± 0.43	**86.33 ± 0.53**	**81.57 ± 0.58**	10.00 ± 0.00	10.00 ± 0.00	10.00 ± 0.00
	Uniform	86.24 ± 1.02	78.87 ± 0.66	74.04 ± 2.14	68.29 ± 1.20	48.85 ± 0.58	31.68 ± 1.26	10.00 ± 0.00	10.00 ± 0.00	10.00 ± 0.00
	Lamp	89.56 ± 1.60	88.70 ± 1.71	87.37 ± 1.03	86.98 ± 0.60	82.48 ± 0.94	73.91 ± 2.38	29.92 ± 1.12	16.46 ± 2.63	10.00 ± 0.00
	Regularized	**91.33 ± 0.21**	**90.69 ± 0.24**	**89.94 ± 0.16**	**89.02 ± 0.26**	86.16 ± 0.30	80.56 ± 0.23	**43.07 ± 3.47**	**20.94 ± 2.77**	10.00 ± 0.00
Inception-V3	Global	94.51 ± 0.17	94.65 ± 0.12	94.30 ± 0.19	**94.04 ± 0.16**	**93.18 ± 0.12**	**91.44 ± 0.13**	51.17 ± 3.95	18.45 ± 5.14	10.00 ± 0.00
	Uniform	94.01 ± 0.17	93.38 ± 0.20	92.32 ± 0.35	90.14 ± 0.20	84.72 ± 1.71	61.76 ± 4.99	10.00 ± 0.00	10.00 ± 0.00	10.00 ± 0.00
	Lamp	94.41 ± 0.14	94.26 ± 0.10	94.34 ± 0.06	93.85 ± 0.19	92.58 ± 0.24	90.84 ± 0.10	54.40 ± 1.46	26.65 ± 2.56	10.00 ± 0.00
	Regularized	**94.81 ± 0.14**	**94.68 ± 0.16**	94.34 ± 0.21	93.90 ± 0.11	92.89 ± 0.15	90.93 ± 0.10	**66.63 ± 0.40**	**38.02 ± 3.76**	10.00 ± 0.00
Mobilenet-V2	Global	84.19 ± 0.42	81.50 ± 0.68	72.04 ± 2.00	50.48 ± 5.20	11.80 ± 3.61	10.00 ± 0.00	10.00 ± 0.00	10.00 ± 0.00	10.00 ± 0.00
	Uniform	64.91 ± 0.77	34.33 ± 1.36	18.65 ± 10.59	10.00 ± 0.00	10.00 ± 0.00	10.00 ± 0.00	10.00 ± 0.00	10.00 ± 0.00	10.00 ± 0.00
	Lamp	84.77 ± 0.47	83.52 ± 0.30	81.48 ± 0.62	77.30 ± 0.48	58.53 ± 0.83	28.67 ± 0.93	10.00 ± 0.00	10.00 ± 0.00	10.00 ± 0.00
	Regularized	**86.85 ± 0.29**	**86.35 ± 0.33**	**84.60 ± 0.34**	**81.90 ± 0.32**	**71.95 ± 0.30**	**42.07 ± 2.91**	10.00 ± 0.00	10.00 ± 0.00	10.00 ± 0.00
ResNet-18	Global	91.27 ± 0.15	91.07 ± 0.16	90.89 ± 0.14	90.29 ± 0.10	89.04 ± 0.21	87.29 ± 0.25	74.07 ± 0.71	58.74 ± 0.63	17.72 ± 1.25
	Uniform	90.36 ± 0.15	88.99 ± 0.21	87.81 ± 0.24	86.05 ± 0.24	78.69 ± 1.06	68.27 ± 1.57	16.14 ± 3.15	12.81 ± 2.52	10.00 ± 0.00
	Lamp	91.39 ± 0.18	**91.38 ± 0.15**	90.90 ± 0.15	90.44 ± 0.14	89.05 ± 0.23	86.80 ± 0.27	72.70 ± 0.37	57.28 ± 0.67	18.62 ± 2.25
	Regularized	**91.56 ± 0.24**	91.35 ± 0.10	**91.05 ± 0.22**	**90.66 ± 0.23**	**89.28 ± 0.19**	**87.40 ± 0.32**	**74.35 ± 0.54**	**63.16 ± 0.27**	**24.13 ± 2.48**
VGG-16	Global	88.37 ± 0.82	43.90 ± 22.30	18.62 ± 5.25	19.66 ± 6.39	14.77 ± 6.07	12.85 ± 3.49	10.00 ± 0.00	10.00 ± 0.00	10.00 ± 0.00
	Uniform	88.88 ± 0.54	78.44 ± 0.38	64.14 ± 1.08	49.72 ± 1.16	26.48 ± 2.81	12.23 ± 2.09	10.24 ± 0.49	10.00 ± 0.00	10.00 ± 0.00
	Lamp	92.57 ± 0.16	**90.55 ± 0.29**	86.84 ± 0.63	80.99 ± 0.52	67.93 ± 0.35	49.24 ± 0.11	13.74 ± 2.21	10.47 ± 0.65	9.98 ± 0.04
	Regularized	**92.61 ± 0.18**	90.49 ± 0.37	**87.27 ± 0.31**	**81.72 ± 0.17**	**68.70 ± 0.52**	**53.43 ± 0.59**	**16.73 ± 2.68**	10.36 ± 0.62	10.11 ± 0.20
VGG16-BN	Global	92.70 ± 0.34	92.76 ± 0.15	92.90 ± 0.35	92.60 ± 0.38	92.13 ± 0.15	**91.24 ± 0.21**	82.11 ± 3.12	65.65 ± 3.33	11.09 ± 2.18
	Uniform	92.48 ± 0.19	90.38 ± 3.25	91.04 ± 0.29	89.49 ± 0.22	85.78 ± 3.28	76.72 ± 5.11	13.47 ± 4.53	11.07 ± 2.14	10.00 ± 0.00
	Lamp	92.63 ± 0.16	92.71 ± 0.38	92.55 ± 0.23	92.43 ± 0.14	92.03 ± 0.31	90.75 ± 0.22	83.11 ± 0.45	72.21 ± 0.25	29.46 ± 3.38
	Regularized	**93.37 ± 0.25**	**93.67 ± 0.19**	**93.34 ± 0.24**	**93.15 ± 0.19**	**92.42 ± 0.14**	91.19 ± 0.14	**83.44 ± 0.23**	**73.87 ± 0.69**	**38.89 ± 1.41**

A.4 Parallelism between Kullback-Leibler divergence and R_{out}

Assuming output perturbance changes Y (which is considered as a random variable) to Y_d with a scaling factor R_{out}, the Kullback-Leibler divergence (KL) between Y_d and Y can be calculated as follows [1]:

$$Y_d = R_{out} Y \tag{A.1}$$

The PDF of Y_d becomes (shown by Bertsekas and Tsitsiklis [54]):

$$f_{Y_d}(y_d) = f_Y\left(\frac{y_d}{R_{out}}\right)\left|\frac{1}{R_{out}}\right| \tag{A.2}$$

The KL divergence $D_{KL}(Y \parallel Y_d)$ is given by:

$$D_{KL}(Y \parallel Y_d) = \int_{-\infty}^{\infty} f_Y(y) \log\left(\frac{f_Y(y)}{f_{Y_d}(y)}\right) dy \tag{A.3}$$

Substituting the expression for $f_{Y_d}(y)$:

$$f_{Y_d}(y) = f_Y\left(\frac{y}{R_{out}}\right)\left|\frac{1}{R_{out}}\right| \tag{A.4}$$

$$D_{KL}(Y \parallel Y_d) = \int_{-\infty}^{\infty} f_Y(y) \log\left(\frac{f_Y(y)}{f_Y\left(\frac{y}{R_{out}}\right)} \cdot R_{out}\right) dy \tag{A.5}$$

Breaking down the logarithm:

$$D_{KL}(Y \parallel Y_d) = \int_{-\infty}^{\infty} f_Y(y) \left(\log\left(\frac{f_Y(y)}{f_Y\left(\frac{y}{R_{out}}\right)}\right) + \log(R_{out})\right) dy \tag{A.6}$$

$$D_{KL}(Y \parallel Y_d) = \int_{-\infty}^{\infty} f_Y(y) \log\left(\frac{f_Y(y)}{f_Y\left(\frac{y}{R_{out}}\right)}\right) dy + \int_{-\infty}^{\infty} f_Y(y) \log(R_{out})\, dy \tag{A.7}$$

[1] Partial help from GPT-4 was used for all derivations written in this section

The second term simplifies since $\log(R_{out})$ is a constant and $\int_{-\infty}^{\infty} f_Y(y)\,dy = 1$:

$$\int_{-\infty}^{\infty} f_Y(y) \log(R_{out})\,dy = \log(R_{out}) \tag{A.8}$$

Therefore, the KL divergence $D_{KL}(Y \parallel Y_d)$ becomes:

$$D_{KL}(Y \parallel Y_d) = \int_{-\infty}^{\infty} f_Y(y) \log\left(\frac{f_Y(y)}{f_Y\left(\frac{y}{R_{out}}\right)}\right) dy + \log(R_{out}) \tag{A.9}$$

Considering the terms $f_Y(y)$ and $\log\left(\frac{f_Y(y)}{f_Y\left(\frac{y}{R_{out}}\right)}\right)$ are intractable or hard to estimate, $\log(R_{out})$ term is aligned with the divergence term. Therefore, we can say that if R_{out} term increases, divergence increases.

A.5 Relationship between τ and learning rate decay in GPT-2 model

To further investigate the relationship between the learning rate and pruning speed (τ), with GPT-2 model, we examine the effect of learning rate on the results. The results show that if τ is slow (pruning is distributed over the training steps) and learning decay is fast (i.e., the learning rate becomes too slow in early steps), small learning rate can not catch up with the high pruning speed towards the end of training steps. When τ is high (leading to faster pruning and reaching P_{final} at the very early stages), this effect is observed relatively less. This causes the model accuracy to degrade with high learning rate decay and small τ. To align learning rate decay with the pruning, we add a proportionate decay slowing term for cosine annealing [55] term as in Eqn. A.10 where τ is exponential pruning schedule decay parameter, η's represent learning rates predetermined constants and T is the total or current training steps.

$$\eta_t = \eta_{min} + \frac{1}{2}(\eta_{max} - \eta_{min})(1 + \cos(\frac{T_{cur}}{T_{max}}\pi))^{\frac{\tau}{10}} \tag{A.10}$$

Note on Choice of Pruning Parameter (τ): τ is a parameter that determines the proportion of model pruning at a given step, as explained in Section 2.2.4. Since τ governs the aggressiveness of pruning and the ratio of sparsity at a given training step t, it is independent of the model size and can be considered a generic control for pruning aggressiveness.

If the pruning is not aggressive (i.e., τ is small), the model keeps significant number of weights for the majority of the training process (e.g., more than 50% of the total training steps). This results in a prolonged period of training with relatively less loss due to distributed weight removal. However, continued pruning towards the end of training process may result in increased loss. Conversely, with a high τ value, the pruning process becomes overly aggressive, quickly reducing the model's search space before sufficient useful information is captured during training. This early reduction damages the model's generalization capability.

Through our investigation of τ versus validation accuracy, as shown in Figure A.5, we select a moderate τ value. This choice balances the benefits of aggressive initial pruning with sufficient flexibility for fine-tuning later in training.

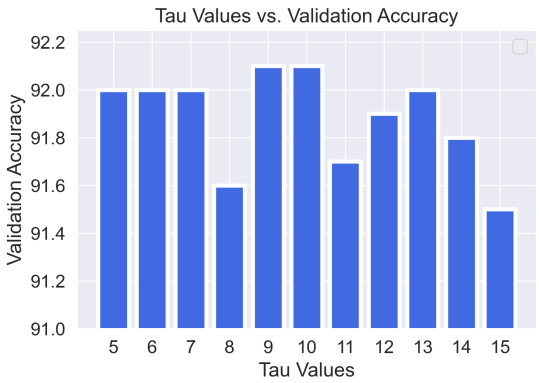


Figure A.5: Tau vs. Test Accuracy for VGG-16 Pruned at 99%

A.6 Results on GPT-2 Model Trained for 12000 Steps

The results do not differ significantly from the models trained for 6000 steps. Thus, we can say that the model converges in 6000 steps and finetuning for more steps is not needed to increase the model performance. The results can be seen in Table A.5

Table A.5: GPT-2 trained for 12000 steps on OpenWeb dataset validation loss

Pruning (%)	Regularized	Wanda
0.95	**6.80**	6.84
0.97	**7.08**	7.59
0.99	**7.43**	9.43

Appendix B

Quantization

B.1 Training Specifications

Similar to previous section, we used PyTorch [31] framework for the pruning-quantization experiments. We have utilized Pytorch implementation of Tensorflow static quantization [41] and used same hyperparameters for comparison with our work. Only difference is in pruning and quantization methods.

B.1.1 Tests with Cifar-10 Dataset

B.1.1.1 Hyperparameters

- Batch Size: 2048
- Total Training Steps: 50 * Steps per epoch
- Total Training Epochs: 50
- Initial Learning Rate: 0.0001 for VGG-16, 0.001 for other models)
- Learning Rate Decay: 0.9 for every 10 epochs

- Tau (For Exponential Decay Pruning): 10

B.1.1.2 Hardware Specifications

- GPU: NVIDIA A100-80GB Tensor Core GPU
- CPU: AMD 7742 2.24 GHz
- RAM: 1TB
- Operating System: Red Hat Enterprise Linux 8.5 (Ootpa)

B.1.2 Tests with Imagenet-200 Dataset

B.1.2.1 Hyperparameters

- Batch Size: 2048
- Total Training Steps: 20 * Steps per epoch
- Total Training Epochs: 20
- Initial Learning Rate: 0.0001
- Tau (For Exponential Decay Pruning): 10

B.1.2.2 Hardware Specifications

- GPU: NVIDIA A100-80GB Tensor Core GPU
- CPU: AMD 7742 2.24 GHz
- RAM: 1TB
- Operating System: Red Hat Enterprise Linux 8.5 (Ootpa)

Appendix C

Logarithmic Hardware

C.1 Model Training Specifications

To train the models and test with validation datasets, we used Tensorflow framework [56]

C.1.1 MNIST Dataset

- Batch Size: 128
- Optimizer: Adam
- Learning Rate: 0.01

C.1.2 CIFAR-10 Dataset

C.1.2.1 ResNet-18

- Batch Size: 128

- Optimizer: Adam
- Learning Rate: 0.001
- Image Upscaling to 64x64
- Epochs: 50
- Model is trained by using randomly initialized weights

C.1.2.2 VGG-16

- Batch Size: 128
- Optimizer: Adam
- Learning Rate: 0.0001
- Transfer learning is applied, weights are pretrained on Imagenet-1000

C.2 Floating Point Unit Design

Verilog code for floating point unit is taken from [53].

C.2.1 Imagenet-1000 Dataset

To run validation dataset on VGG-16 (trained with Imagenet-1000), we used the weights provided by Tensorflow and perform the tests on pretrained VGG-16 model with Imagenet-1000 validation dataset.

C.3 FPGA Utilization and Power Tests Specifications

Vivado Version: 2021.1 FPGA Configuration: AMD Zynq Ultrascale+ MPSoC EV XCZU7 FPGA

C.4 Model Training Hardware and CUDA Simulation Specifications

- CUDA [57] Version: 12.1
- CPU: AMD Ryzen 7 5800X3D 8-Core Processor
- GPU: NVIDIA RTX 2080 SUPER
- Memory: 16 GB

Appendix D

Usage of Generative Artificial Intelligence

We acknowledge the usage of Generative Artificial Intelligence in our work in compliance with Higher Education Council [58] regulation. We disclose the extent where Large Language Models (LLMs) are used within this work.

D.1 Usage of Large Language Models

We used Large Language Models for grammatical soundness of the text and create plots with a given result data format.

Grammar-Text Suggestion: Google Gemini Bard [59], ChatGPT 4 by OpenAI [60].
Translation: ChatGPT 4
Plot Code Generation: Code for Figure 2.4, and 2.5 are mostly generated by ChatGPT4
Mathematical Suggestions for Appendix A.4: ChatGPT4.