The Dissertation Committee for Zixuan Jiang
certifies that this is the approved version of the following dissertation:

Efficient Machine Learning Software Stack from Algorithms to Compilation

Committee:

David Zhigang Pan, Supervisor

Diana Marculescu

Atlas Wang

Qiang Liu

Yuan Yu

Efficient Machine Learning Software Stack from Algorithms to Compilation

by

Zixuan Jiang

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

August 2023

Acknowledgments

I sincerely thank my advisor, Professor David Z. Pan, for his guidance and support over the years. He is a wise leader who appropriately manages research projects and encourages me to pursue meaningful and impactful research problems. He is also a great mentor who guides my research, develops my skills for a future career, and cultivates my attitude toward healthy life. His warm advice and continuous support will positively impact my life in the future.

I would also like to thank other committee members. In particular, I want to thank Dr. Yuan Yu for his help during my internship in the ONNX Runtime team at Microsoft Cloud + AI. As a distinguished expert in machine learning systems, his insights help me understand the issues in the existing solutions. His energetic working style has long-term impacts on my future career and research. I also thank Professor Qiang Liu for his technical inspiration and suggestions on the mixed precision neural architecture search project. His enthusiasm for optimization techniques always encourages me to pursue more effective and elegant solutions. I would also like to express my thankfulness to Professor Diana Marculescu and Professor Atlas Wang for their kindness and support to this dissertation.

Besides, I really appreciate my colleagues during my internships. Dr. Yuan Yu, Sherlock Baihan Huang, and Dr. Wei Zuo were mentors for my internship at Microsoft Cloud + AI. They gave me a lot of precious advice on working with both industry and academia. Dr. Ebrahim M. Songhori, Shen Wang, Dr. Joe Wenjie Jiang, Dr. Azalia Mirhoseini, and

Dr. Anna Goldie were mentors for my internship at Google Research, Brain Team. They offered precious insights and experiences in physical design algorithms and reinforcement learning. Dr. Lan Nie and Dr. Vineet Gupta were mentors for my internship at Google Ads. They taught me how to handle real-world problems with machine learning algorithms and infrastructures. Dr. Lifeng Nai, Dr. Safeen Huda, Dr. Sheng Li, and Dr. Jishen Zhao were mentors for my internship in the Google TPU team. They provided various domain-specific insights regarding machine learning systems and accelerators.

In addition to my advisor, committee members, and colleagues, I am lucky to work and collaborate with many other people: Chengyue Gong, Dilin Wang at Computer Science Department of UT Austin, UTDA members and alums, including Dr. Yibo Lin, Dr. Meng Li, Dr. Biying Xu, Dr. Wuxi Li, Dr. Shounak Dhar, Dr. Zheng Zhao, Dr. Wei Ye, Dr. Mohamed Baker Alawieh, Dr. Wei Shi, Dr. Keren Zhu, Dr. Mingjie Liu, Dr. Jiaqi Gu, Dr. Chenghao Feng, Dr. Hao Chen, Ahmet F. Budak, Rachel S. Rajarathnam, Hanqing Zhu, Hyunsu Chae, Chen-Hao Hsu, et al. Their kindness in inspiring discussion and efforts for productive collaborations help develop and polish this dissertation.

Last, but not least, I am deeply thankful to my family. Without their support and sacrifice, it would have been impossible for me to finish this dissertation and pursue the PhD degree at UT Austin.

Efficient Machine Learning Software Stack from Algorithms to Compilation

Publication No. _____

Zixuan Jiang, Ph.D.
The University of Texas at Austin, 2023

Supervisor: David Zhigang Pan

Machine learning enables the extraction of knowledge from data and decision-making without explicit programming, achieving great success and revolutionizing many fields. These successes can be attributed to the continuous advancements in machine learning software and hardware, which have expanded the boundaries and facilitated breakthroughs in diverse applications.

The *machine learning software stack* is a comprehensive collection of components used to solve problems with machine learning algorithms. It encompasses problem definitions, data processing, model and method designs, software frameworks, libraries, code optimization, and system management. This stack supports the entire life cycle of a machine learning project. The software stack allows the community to stand on the shoulders of previous great work and push the limit of machine learning, fostering innovation and enabling broader adoption of machine learning techniques in academia and industry.

The software stack is usually divided into *algorithm* and *compilation* with distinct design principles. Algorithm design prioritizes task-related performance, while compilation

focuses on execution time and resource consumption on hardware devices. Maintaining arithmetic equivalence is optional in algorithm design, but compulsory in compilation to ensure consistent results. The compilation is closer to hardware than algorithm design. Compilation engineers optimize for hardware specifications, while algorithm developers usually do not prioritize hardware-friendliness. Opportunities to enhance hardware efficiency exist in algorithm and compilation designs, as well as their interplay.

Despite extensive innovations and improvements, efficiency in the machine learning software stack is a continuing challenge. Algorithm design proposes efficient model architectures and learning algorithms, while compilation design optimizes computation graphs and simplifies operations. However, there is still a gap between the demand for efficiency and the current solutions, driven by rapidly growing workloads, limited resources in specific machine learning applications, and the need for cross-layer design. Addressing these challenges requires interdisciplinary research and collaboration. Improving efficiency in the machine learning software stack will optimize performance and enhance the accessibility and applicability of machine learning technologies.

In this dissertation, we focus on addressing these efficiency challenges from the perspectives of machine learning algorithms and compilation.

We introduce three novel improvements that enhance the efficiency of mainstream machine learning algorithms. Firstly, *effective gradient matching for dataset condensation* generates a small insightful dataset, accelerating training and other related tasks. Additionally, *NormSoftmax* proposes to append a normalization layer to achieve fast and stable training in Transformers and classification models. Lastly, *mixed precision hardware-aware neural architecture search* combines mixed-precision quantization, neural architecture search,

and hardware energy efficiency, resulting in significantly more efficient neural networks than using a single method.

However, algorithmic efficiency alone is insufficient to fully exploit the potential in the machine learning software stack. We delve into and optimize the compilation processes with three techniques. Firstly, we simplify the layer normalization in the influential Transformers, obtaining *two equivalent and efficient Transformer variants* with alternative normalization types. Our proposed variants enable efficient training and inference of popular models like GPT and ViT. Secondly, we formulate and solve *the scheduling problem for reversible neural architectures*, finding the optimal training schedule that fully leverages the computation and memory resources on hardware accelerators. Lastly, *optimizer fusion* allows users to accelerate the training process in the eager execution mode of machine learning frameworks. It leverages the better locality on hardware and parallelism in the computation graphs.

Throughout the dissertation, we emphasize the integration of efficient algorithms and compilation into a cohesive machine learning software stack. We also consider hardware properties to provide hardware-friendly software designs.

We demonstrate the effectiveness of the proposed methods in algorithm and compilation through extensive experiments. Our approaches effectively reduce the time and energy required for both training and inference. Ultimately, our methods have the potential to empower machine learning practitioners and researchers to build more efficient, powerful, robust, scalable, and accessible machine learning solutions.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Machine learning, an ever-evolving field of artificial intelligence, is dedicated to the extraction of knowledge from data and the ability to make predictions or decisions without explicit programming. Its emergence as a versatile and potent methodology has captivated widespread attention, particularly through the advancements in deep learning, thereby revolutionizing how we analyze data and derive meaningful insights [104].

At its core, machine learning harnesses statistical techniques and computational power to enable *machines* to *learn*. By analyzing and discovering patterns in extensive datasets, machine learning can unveil valuable insights [21, 148], achieve remarkable prediction accuracy [63, 183], and automate decision-making processes [166].

The success of machine learning can be attributed to the continuous development and innovation in both software and hardware technologies, as well as their synergistic evolution. The continuous development of machine learning software and the availability of powerful hardware accelerators have allowed researchers and practitioners to push the boundaries of what is achievable in the realm of machine learning. As these technologies continue to advance in tandem, we can expect even more remarkable breakthroughs and broader applications of machine learning in various domains.

Layer	Example	Stratum	Equivalence
ML Problems	Machine Translation	ML Algorithm	Unnecessary
ML Datasets	Wikipedia		
ML Models	Transformer		
ML Methods	Pretraining and Finetuning		
ML Frameworks	TensorFlow	ML Compilation	Guaranteed
ML Compilers	XLA		
ML Libraries	CUDA		
Operating Systems	Linux		

Figure 1.1: Machine learning (ML) software stack, adapted from [60].

1.1 Machine Learning Software Stack

The *machine learning software stack* encompasses the entire software pipeline used to solve problems with machine learning techniques. It consists of a comprehensive collection of components, including problem definitions, data processing and engineering, model and method designs, software frameworks and tools, libraries, code generation and optimization, and system management. These components are utilized throughout the development, training, evaluation, and deployment of machine learning models, as depicted in Figure 1.1.

The machine learning software stack comprises multiple layers, which can be illustrated using an example. Suppose we aim to address the machine translation problem [41] using machine learning. Initially, we formulate it as a sequence-to-sequence problem and utilize Wikipedia pages as our corpus. We then employ self-supervised pre-training and supervised fine-tuning on Transformer-based models [183]. The algorithm is implemented in the TensorFlow framework [1], described as a computation graph. The computation graph is then compiled using the XLA compiler [159] and linked to the CUDA library [50]. Lastly,

the workload and related hardware resources are managed by the Linux operating system.

A well-designed machine learning software stack provides a robust and efficient workflow that supports the entire life cycle of a machine learning project. This enables researchers and practitioners to experiment with different algorithms, optimize model performance, handle large-scale datasets, and seamlessly deploy models into production systems.

Furthermore, the machine learning software stack promotes collaboration and knowledge sharing within the community. Researchers and practitioners can leverage existing libraries and tools, build upon each other's work, and actively contribute to open-source projects. This fosters innovation and accelerates advancements in the field. Additionally, the availability of comprehensive documentation, tutorials, and online resources facilitates the learning and adoption of machine learning techniques.

The collective progress across all layers in the software stack is instrumental in the success of machine learning. An efficient, scalable, and robust machine learning software stack is fundamental and indispensable for future machine learning solutions. As the field continues to advance, the machine learning software stack will play a pivotal role in democratizing access to machine learning, enabling individuals and organizations to harness the power of data-driven decision-making.

1.2 Machine Learning Algorithms and Compilation

The software stack is typically divided into two categories: *algorithm* and *compilation*. Algorithm design focuses on approaching a problem using machine learning, while compilation involves mapping the workload to various hardware platforms, ranging from CPUs and

GPUs to specialized accelerators. There are three notable design principles that differentiate algorithm design from compilation design.

Priority. In algorithm design, the top priority is task-related performance, such as translation quality and classification accuracy. Researchers strive to improve the quality of results by formulating the problem, processing the dataset, and proposing new models and methods. On the other hand, compilation design emphasizes execution time and resource consumption on real hardware devices. The focus is on the speed and cost of completing a machine learning workload.

Arithmetic Equivalence. Maintaining arithmetic equivalence is optional when proposing a new machine learning algorithm since the goal is to achieve higher quality on specific problems. Guaranteeing equivalence limits the potential for better performance. On the contrary, machine learning compilation guarantees the same arithmetic functionality of the workload. A new compilation method must always maintain mathematical equivalence to ensure consistent results.

Distance to Hardware. In comparison to machine learning algorithms, the compilation is inherently closer to the hardware. While algorithm developers often do not prioritize hardware-friendliness, compilation engineers are tasked with considering and optimizing against hardware specifications. They usually navigate the intricacies of the underlying hardware architecture to extract maximum performance and efficiency. Both algorithm and compilation designs offer opportunities to enhance hardware efficiency. Recognizing the mutual influence and interplay between algorithms and compilation allows us to further unlock the full potential of hardware efficiency.

Above all, the substantial differences between machine learning algorithms and compilation give rise to distinct design philosophies in these two areas. This is why we address them in separate chapters in this dissertation.

1.3 Efficiency as a Continuing Challenge in Software Stack

Extensive innovations and improvements are being made to enhance efficiency in the machine learning software stack. In algorithm design, there are proposals for efficient model architectures [95, 161, 177] and learning algorithms [92, 121]. In compilation design, the community works on optimizing computation graphs and simplifying operations [219]. Notably, several efficient machine learning frameworks, such as TensorFlow [1], PyTorch [142], and MxNet [31], have emerged and matured, becoming critical tools for the community.

However, despite these advancements, there still exists a gap between the demand for efficiency and the current machine learning solutions. As the need for more sophisticated machine learning solutions continues to grow, there is an increasing requirement for an efficient and optimized software stack that covers the entire machine learning workflow, from algorithmic development to compilation processes. We have identified several critical challenges in achieving efficiency within this stack.

Scalability to large workloads. The rapidly escalating workloads continuously challenge the efficiency of machine learning solutions. Model size and data volume have grown expeditiously [88]. Large models, especially large language models (LLMs) [217] and vision models [42], have raised higher demands for the entire stack, including data preprocessing and management, distributed training, model inference, etc.

Tiny workloads with limited resources. With advancements in technologies such as the Internet of Things (IoT) and edge computing, it is desirable to incorporate machine learning techniques into resource-constrained embedded devices for distributed and ubiquitous intelligence [90]. Specific applications, like autonomous vehicles and edge devices, have strict efficiency, latency, and bandwidth constraints. These applications require the development of more efficient machine learning solutions tailored to these resource-constrained environments.

Cross-layer design. In addition to improving efficiency in each individual layer, a significant challenge lies in the lack of cross-layer design, where the interplay between algorithms, compilation, and hardware is not fully exploited. Considering the entire software stack as a unified system can lead to more efficient and effective solutions.

Addressing these challenges requires interdisciplinary research efforts and collaboration across multiple domains. By developing efficient machine learning solutions, significant advancements can be made in various fields, enabling the deployment of intelligent systems in both large-scale systems and resource-constrained environments. Efforts to improve efficiency within the machine learning software stack will optimize performance and expand the accessibility and applicability of machine learning technologies.

1.4 Contributions and Overview of the Dissertation

This dissertation addresses the challenges associated with developing an efficient machine learning software stack and exploring innovative techniques to streamline the entire process. Figure 1.2 provides a visual representation of the hierarchical overview of the dissertation, illustrating the different components and their interconnections. We propose a

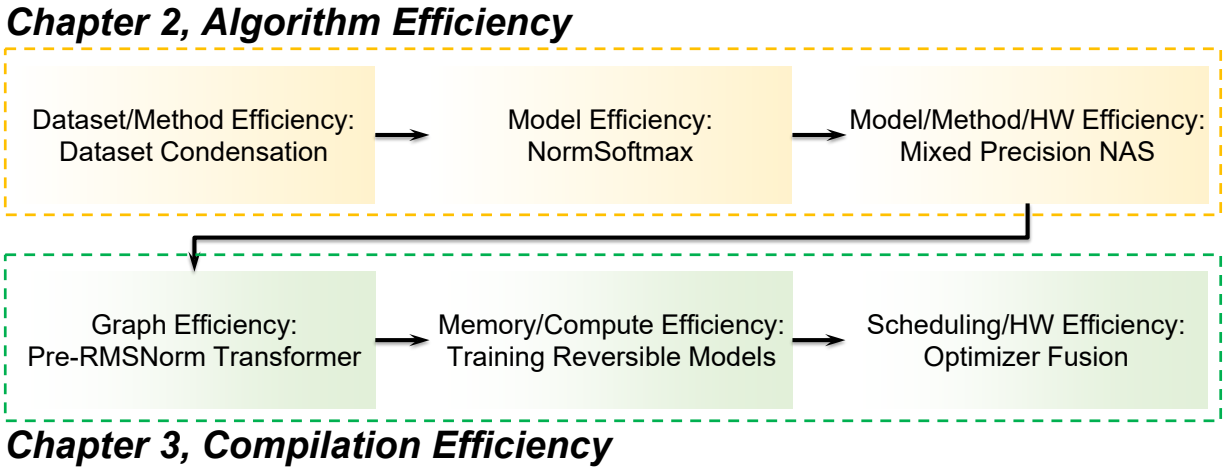


Figure 1.2: The overview of the dissertation.

cross-layer synergistic design to enhance machine learning efficiency in the perspectives of algorithms and compilation. Additionally, we incorporate hardware awareness and friendliness to further improve the efficiency of our proposed methods.

Chapter 2 presents our efficiency improvement in machine learning algorithms. Our improvements are in various layers, including datasets, models, and methods. This chapter showcases our representative work, listed below.

- **Effective Gradient Matching for Dataset Condensation.** We delve into and improve the gradient matching algorithm to condense datasets. We enhance the efficiency and effectiveness of machine learning datasets.
- **NormSoftmax: Normalizing Softmax Input to Accelerate and Stabilize Training.** We propose the addition of a normalization layer to mainstream neural architectures, such as Transformers or cross-entropy loss. This appended normalization improves the efficiency and robustness when training softmax-based models.

- **Mixed Precision Hardware-Aware Neural Architecture Search.** We propose a framework that combines neural architecture search, mixed precision quantization, and hardware energy efficiency. This integrated approach enhances the efficiency of models, search algorithms, and hardware deployment.

However, exploiting the full potential of the machine learning software stack requires more than just algorithmic efficiency.

Chapter 3 delves into the compilation processes of transforming high-level machine learning models into efficient computation graphs and schedules. By studying advanced compilation strategies such as computation graph simplification, scheduling, and memory management, we aim to fully harness the computational capabilities of modern hardware platforms and achieve significant performance improvements. Our contributions in this chapter are highlighted as follows.

- **Pre-RMSNorm and Pre-CRMSNorm Transformers: Equivalent and Efficient Pre-LN Transformers.** We unify two widely used normalization types in Transformers and propose two more equivalent Transformer variants. We improve the efficiency of computation graphs for Pre-LN Transformers, such as the impactful GPT [21, 148] and ViT [47].
- **An Efficient Training Framework for Reversible Neural Architectures.** We formulate the scheduling problem for reversible neural architectures and propose an algorithm to obtain optimal schedules. We improve memory and computation efficiency when training on invertible neural networks.

- **Optimizer Fusion: Efficient Training with Better Locality and Parallelism.** We propose to fuse the optimizer with backward or forward computations during the training process, enhancing the efficiency of machine learning frameworks by improving locality and parallelism.

Throughout this dissertation, we emphasize the importance of an integrated and cohesive machine learning software stack by seamlessly incorporating efficient algorithms and compilation designs. Additionally, we consider hardware properties to provide hardware-aware and hardware-friendly software designs.

Chapter 4 concludes the dissertation by summarizing the key findings and suggesting several future directions for further exploration in improving the machine learning software stack.

This dissertation aims to contribute to developing an efficient machine learning software stack by addressing efficiency challenges in both algorithm and compilation designs. The outcomes of this research are expected to advance the field of machine learning, empowering practitioners and researchers to build more efficient, powerful, robust, scalable, and accessible machine learning solutions in this rapidly evolving field.

Chapter 2

Efficient Machine Learning Algorithms

Efficiency in machine learning algorithms refers to the ability of an algorithm to achieve high performance while minimizing data resources, computational resources, time requirements, etc. Most algorithms seek a balance between efficiency and task-related performance, such as classification accuracy.

For instance, few-shot learning [169] is a method that achieves this balance by working with limited data points. It focuses on data efficiency while maintaining satisfactory task-related performance. Pruning, quantization [113], and other approximation methods can be employed to trade off task-related performance for reduced computational requirements,

---

This chapter is based on the following publications.

1. Chengyue Gong*, Zixuan Jiang*, Dilin Wang, Yibo Lin, Qiang Liu, David Z Pan. "Mixed Precision Neural Architecture Search for Energy Efficient Deep Learning". IEEE/ACM International Conference on Computer-Aided Design (ICCAD), Westminster, CO, USA, 2019 [57].
2. Zixuan Jiang, Jiaqi Gu, Mingjie Liu, David Z Pan. "Delving into effective gradient matching for dataset condensation". IEEE International Conference on Omni-layer Intelligent Systems (COINS), Berlin, Germany, 2023 [83].
3. Zixuan Jiang, Jiaqi Gu, David Z Pan. "NormSoftmax: Normalizing the Input of Softmax to Accelerate and Stabilize Training". IEEE International Conference on Omni-layer Intelligent Systems (COINS), Berlin, Germany, 2023 [85].

For the first publication, Chengyue Gong and I shared equal contributions. I took charge of the hardware-related algorithms and experiments. For the other two publications, I am the main contributor in charge of problem formulation, algorithm development, and experimental validations.

making them suitable for scenarios where efficiency is of utmost importance.

In this chapter, we present three methods. The first work, effective gradient matching for dataset condensation, enables us to achieve higher data efficiency. The second approach, NormSoftmax, accelerates the training process of softmax-based neural networks. We simplify the training at the expense of little degradation of model representability. The third method, mixed precision quantization neural architecture search, pushes the Pareto front for the quantization technique. It makes a trade-off between computational efficiency and task-related performance.

2.1 Delving into Effective Gradient Matching for Dataset Condensation

Large datasets are critical for the success of deep learning at the cost of computation and memory. The high cost is unbearable when we train deep learning models with limited training time or memory budget. For example, quick training is needed when training is a subtask. When we perform neural architecture search [49], hyper-parameter optimization [11, 52], or training algorithm design and validation, we expect to obtain training performance quickly. Another example is the training with limited storage space. To overcome catastrophic forgetting in continual learning, we usually save partial samples for future training [94]. There is a harsh constraint on the memory space when training on edge devices [187]. Above all, it is a critical problem to achieve data efficiency in deep learning training.

As a traditional method to reduce the size of the training dataset, coreset construction defines a criterion for representativeness [3, 26, 51, 153, 163] and then selects samples based

on the criterion. Coreset construction is used in many efficient and quick training tasks, e.g., accelerating hyperparameter search [165], continual learning [18]. Unlike the coreset construction method, dataset synthesis generates a small dataset, which is directly optimized for the downstream task. Since it does not rely on representative samples, the dataset synthesis outperforms the coreset construction in the corresponding downstream task.

Wang *et al*. formulate the network parameters as a function of the synthetic training set and formulate the dataset condensation task as a bi-level optimization problem [186]. Specifically, the ultimate target is to train deep learning models on the synthetic training set from scratch such that the trained model can generalize to the original training dataset. The authors minimize the training loss on the original large training data by optimizing the synthetic data. Based on the formulation of the bi-level optimization problem, Sucholutsky and Schonlau extend the method by distilling both input and their soft labels [173]. Such *et al*. propose to learn a generative teaching network, which generates synthetic data for training student networks [172]. Nguyen *et al*. use kernel ridge-regression to compress training datasets, enhancing the dataset distillation method [139].

Zhao *et al*. propose to match gradients w.r.t. parameters when training examples come from synthetic and original datasets, respectively, to solve the bi-level optimization problem [216]. This method mimics the first-order loss landscape when the real training set is used and intuitively maximizes the landscape similarity via gradient matching. By directly targeting the training dynamics, this optimization-aware methodology achieves the current state-of-the-art performance on dataset condensation. However, the previous method does not deeply investigate the working principle in gradient matching, and the current matching flow has limited effectiveness and learning efficiency.

2.1.1 Overview

We analyze the gradient matching method from a comprehensive perspective, including what, how, and where to match. We enhance the gradient matching algorithm with three essential techniques to achieve higher efficiency and better task-specific performance. We highlight our contributions as follows.

- *Multi-level matching.* We jointly explore intra-class and inter-class gradient matching to improve performance without extra gradient computation.
- *Overfitting delaying.* We propose to adopt a new type of gradient matching function to mitigate the overfitting issue on the synthetic training set to facilitate the optimization. We concentrate on the angle between the gradients, considering the magnitude simultaneously.
- *Adaptive learning.* We update the synthetic dataset against the parameter where overfitting happens. Thus, we can achieve the same performance with fewer parameter updates, improving the efficiency of the dataset condensation algorithm.

Our implementation is available on GitHub. [1]

2.1.2 Background

In this section, we first introduce the background of dataset condensation. Then we describe the working principle of the gradient matching method as we will analyze and extend it in Section 2.1.3. Similar to the previous work, we take the classification task with

[1] https://github.com/ZixuanJiang/improved-dataset-condensation

balanced class distribution as an example. The algorithm can be easily extended to other problems.

Dataset condensation. Dataset condensation is a task to generate a small synthetic training dataset $\mathcal{S}$ to mimic the model optimization behavior with the original training set $\mathcal{T}$. Specifically, the network parameter θ is formulated as a function of the synthetic training set $\mathcal{S}$ [186].

$$\theta(\mathcal{S}) = \underset{\theta}{\operatorname{argmin}}\, \mathcal{L}(\mathcal{S}, \theta) \tag{2.1}$$

$\mathcal{L}(\mathcal{S}, \theta) = \frac{1}{|\mathcal{S}|} \sum_{(x,y)\in\mathcal{S}} \ell(f_\theta(x), y)$ is the loss with synthetic dataset $\mathcal{S}$ and model parameter θ. ℓ is the task specific loss function, such as cross entropy loss in the classification task. f_θ represents a deep learning model with parameter θ. Thus, the dataset synthesis problem can be written as the following bi-level optimization problem.

$$\min_{\mathcal{S}} \mathcal{L}\big(\mathcal{T}, \theta(\mathcal{S})\big) \quad \text{s.t. } \theta(\mathcal{S}) = \underset{\theta}{\operatorname{argmin}}\, \mathcal{L}(\mathcal{S}, \theta) \tag{2.2}$$

We train a deep learning model f using the synthetic training set $\mathcal{S}$ from scratch and obtain the optimal parameter $\theta(\mathcal{S})$. The objective is to minimize $\mathcal{L}\big(\mathcal{T}, \theta(\mathcal{S})\big)$, the loss on the original large training set $\mathcal{T}$. In other words, the original dataset $\mathcal{T}$ is the test dataset to verify the model $f_{\theta(\mathcal{S})}$.

Gradient matching algorithm. Among all prior work on dataset condensation, the state-of-the-art performance has been achieved by gradient matching [216], which directly encourages the training dynamics on the synthetic set to mimic that on the real training set. The distance between gradients $\nabla_{\theta_t}\mathcal{L}(\mathcal{S}_t, \theta_t)$ and $\nabla_{\theta_t}\mathcal{L}(\mathcal{T}, \theta_t)$ is minimized, where $t = 0, ..., T$ is the time step. If the gradients match, the training trajectories will be the same using gradient-based optimization methods.

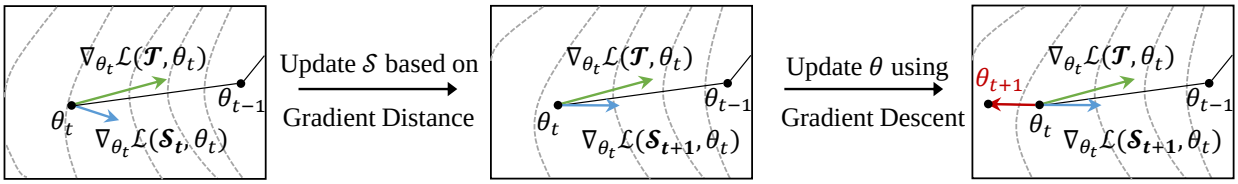


Figure 2.1: One simplified inner loop iteration of dataset condensation with gradient matching algorithm. We first update $\mathcal{S}$ to minimize the gradient distance. Then the network parameter is updated to imitate the training process.

Algorithm 1 shows the original gradient matching method. For each iteration of the outer loop, the model parameters θ_0 are initialized following the distribution of P_{θ_0}. Lines $4 \sim 16$ correspond one inner loop iteration, which is visualized in Figure 2.1. The mini-batches are sampled in the same class when calculating the distance between these two gradients $\nabla_{\theta_t}\mathcal{L}(\mathcal{S}_t,\theta_t)$ and $\nabla_{\theta_t}\mathcal{L}(\mathcal{T},\theta_t)$. Afterwards, the distances for C classes are accumulated to update the synthetic set,

$$\mathcal{S}_{t+1} = \mathcal{S}_t - \eta_{\mathcal{S}}\nabla_{\mathcal{S}_t} D\big(\nabla_{\theta_t}\mathcal{L}(\mathcal{S}_t,\theta_t), \nabla_{\theta_t}\mathcal{L}(\mathcal{T},\theta_t)\big) \tag{2.3}$$

where $D(a,b)$ is a function to measure the distance between two tensors, $\eta_{\mathcal{S}}$ is the learning rate.

In Lines $11 \sim 15$, the parameter θ_t is updated for ζ_θ times to imitate the training process. The gradient $\nabla_{\theta_t}\mathcal{L}(\mathcal{S}_{t+1},\theta_t)$ instead of $\nabla_{\theta_t}\mathcal{L}(\mathcal{S}_t,\theta_t)$ or $\nabla_{\theta_t}\mathcal{L}(\mathcal{T},\theta_t)$ is used to mimic the real training step to update the synthetic set. The red arrow in Figure 2.1 represents a gradient descent step.

The gradient matching method has several extensions. Zhao and Bilen extend it with differentiable data augmentation [215]. Wiewel and Yang propose to learn a weighted combination of shared components to increase memory efficiency [191]. These extensions are

orthogonal to our analysis and enhancements so that our method can be easily integrated into these extensions.

Algorithm 1 Gradient matching algorithm. Our proposed methods are highlighted with color.

Input: Original training set $\mathcal{T}$
Output: Synthetic training set $\mathcal{S}$

Initialize $\mathcal{S}_0$ following Gaussian distribution
for $k = 0, 1, ..., K-1$ **do** ▷ outer loop: explore with different initialization
  Initialize model parameters $\theta_0 \sim P_{\theta_0}$
  for $t = 0, 1, ..., T-1$ **do** ▷ inner loop
    for $c = 0, 1, ..., C-1$ **do**
      Sample mini-batches $B_c^{\mathcal{S}_t} \sim \mathcal{S}_t$, $B_c^{\mathcal{T}} \sim \mathcal{T}$ in the c-th class
      Compute gradients $g_c^{\mathcal{S}_t} = \nabla_{\theta_t}\mathcal{L}(B_c^{\mathcal{S}_t}, \theta_t), g_c^{\mathcal{T}} = \nabla_{\theta_t}\mathcal{L}(B_c^{\mathcal{T}}, \theta_t)$
    end for
    $\texttt{loss} = \sum_{c=0}^{C-1} \mathcal{D}(g_c^{\mathcal{S}_t}, g_c^{\mathcal{T}}) + \lambda\mathcal{D}\left(\frac{1}{C}\sum_{c=0}^{C-1} g_c^{\mathcal{S}_t}, \frac{1}{C}\sum_{c=0}^{C-1} g_c^{\mathcal{T}}\right)$
    $\mathcal{S}_{t+1} = \mathcal{S}_t - \eta_{\mathcal{S}}\nabla_{\mathcal{S}_t}\texttt{loss}$
    $\theta_t^0 = \theta_t$
    for $i = 0, 1, ..., \zeta_\theta(t) - 1$ **do** ▷ update θ with adaptive learning steps
      $\theta_t^{i+1} = \theta_t^i - \eta_\theta\nabla_{\theta_t^i}\mathcal{L}(\mathcal{S}_{t+1}, \theta_t^i)$
    end for
    $\theta_{t+1} = \theta_t^{i+1}$
  end for
end for

2.1.3 Method

In this section, we present our analysis and describe our improvement on the original algorithm. We answer the following questions. *What, how, and where do we match in this gradient matching algorithm?*

2.1.3.1 What We Match: Multi-Level Gradient Matching

The original algorithm matches gradients of the mini-batch that samples in the same class. Specifically, we sample mini-batches $B_c^{\mathcal{S}} \sim \mathcal{S}$, $B_c^{\mathcal{T}} \sim \mathcal{T}$ in the c-th class and calculate the gradients $g_c^{\mathcal{S}} = \nabla_{\theta_t}\mathcal{L}(B_c^{\mathcal{S}}, \theta_t), g_c^{\mathcal{T}} = \nabla_{\theta_t}\mathcal{L}(B_c^{\mathcal{T}}, \theta_t)$, respectively. We minimize the distance between these intra-class gradients with accumulation.

$$\texttt{loss}_{\texttt{intra}} = \sum_{c=0}^{C-1} D(g_c^{\mathcal{S}}, g_c^{\mathcal{T}}) \tag{2.4}$$

Therefore, only the intra-class gradients are matched by using these intra-class mini-batches, missing the inter-class gradient information. However, when we use either $\mathcal{S}$ or $\mathcal{T}$ to train the model, we usually use mini-batches that sample across different classes. To mimic the realistic training process, we also need to match the gradients of these inter-class mini-batches. We propose to match the gradients of these inter-class mini-batches in the following efficient way.

Since $\{g_c^{\mathcal{S}}\}_{c=0}^{C-1}$ has already been computed when calculating the intra-gradient distance, we can directly use them to compute the gradients for the mini-batches $\bigcup_{c=0}^{C-1} B_c^{\mathcal{S}}$ as follows.

$$g_{\cup}^{\mathcal{S}} = \nabla_{\theta_t}\mathcal{L}(\bigcup_{c=0}^{C-1} B_c^{\mathcal{S}}, \theta_t) = \frac{\sum_{c=0}^{C-1} |B_c^{\mathcal{S}}| g_c^{\mathcal{S}}}{\sum_{c=0}^{C-1} |B_c^{\mathcal{S}}|} \tag{2.5}$$

If the mini-batch $B_c^{\mathcal{S}}$ shares the same size, we can further simplify Equation equation 2.5 and obtain $g_{\cup}^{\mathcal{S}} = \frac{1}{C}\sum_{c=0}^{C-1} g_c^{\mathcal{S}}$. We can also assign different weights to different mini-batches $B_c^{\mathcal{S}}$ to mimic the original training set $\mathcal{T}$ if the class distribution is not balanced in $\mathcal{T}$. $g_{\cup}^{\mathcal{T}} = \nabla_{\theta_t}\mathcal{L}(\bigcup_{c=0}^{C-1} B_c^{\mathcal{T}}, \theta_t)$ can be computed in the same way. In this way, we do not perform extra forward and backward computations to calculate gradients for the inter-class mini-batches $\bigcup_{c=0}^{C-1} B_c^{\mathcal{S}}$ and $\bigcup_{c=0}^{C-1} B_c^{\mathcal{T}}$.

With these inter-class gradients, we add a new term in the gradient matching loss as shown in Equation equation 2.6.

$$\underbrace{\sum_{c=0}^{C-1} D(g_c^{\mathcal{S}_t}, g_c^{\mathcal{T}})}_{\texttt{loss}_{\texttt{intra}}} + \lambda \underbrace{D\big(\frac{1}{C}\sum_{c=0}^{C-1} g_c^{\mathcal{S}_t}, \frac{1}{C}\sum_{c=0}^{C-1} g_c^{\mathcal{T}}\big)}_{\texttt{loss}_{\texttt{inter}}} \tag{2.6}$$

The first term is the intra-class gradient matching loss $\texttt{loss}_{\texttt{intra}}$, which is used in the original method. We add a new term of inter-class gradient matching loss, with λ being the weight to balance these two terms. In this multi-level gradient matching loss, we consider both the intra-class and inter-class information. Through experiments, we find that the multi-level gradient matching has better performance than either the intra-class or inter-class counterpart. Experimental results are shown in Section 2.1.4.2.

2.1.3.2 How We Match: Angle and Magnitude

In the original gradient matching algorithm [216], the authors propose to decompose the matching loss layer by layer

$$\mathcal{D}\big(\nabla_\theta \mathcal{L}(\mathcal{S}, \theta), \nabla_\theta \mathcal{L}(\mathcal{T}, \theta)\big) = \sum_{l=1}^{L} d\big(\nabla_{\theta^l} \mathcal{L}(\mathcal{S}, \theta), \nabla_{\theta^l} \mathcal{L}(\mathcal{T}, \theta)\big) \tag{2.7}$$

where L is the number of layers. For each layer, negative cosine similarity is used as the distance between two tensors,

$$d(A, B) = \sum_{i=1}^{\texttt{out}} \big(1 - \frac{A_i \cdot B_i}{\|A_i\| \|B_i\|}\big), \tag{2.8}$$

where `out` is the number of output channels. For example, the weights and the corresponding gradients of a 2D convolution layer has the shape of (`out`, `in/groups`, `h`, `w`) [2]. We reshape

[2] `out` and `in` are the number of output channels and input channels. `groups` is the number of blocked connections from input channels to output channels. `h` and `w` are kernel height and width, respectively.

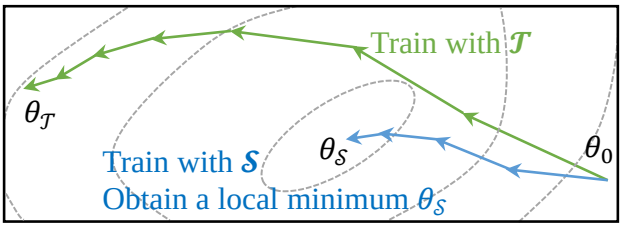


Figure 2.2: Optimization trajectories of training from scratch using $\mathcal{S}$ and $\mathcal{T}$.

the gradients as ($\texttt{out}$, $\texttt{in/groups} \times \texttt{h} \times \texttt{w}$) and compute the cosine similarity for each output channels. This distance function considers the layer-wise structure and output channels, enabling a single learning rate across all layers. By maximizing the cosine similarity between gradients, the $\mathcal{S}$ is expected to lead the parameter in a correct direction.

However, in this distance matching loss, only the angle between gradients is considered, with the magnitude ignored. This is a critical issue when we train a network f from scratch for evaluation using the resultant $\mathcal{S}$. Figure 2.2 visualizes the training process using $\mathcal{S}$ and $\mathcal{T}$ respectively. Since $|\mathcal{S}|$ is usually very small, it is a severe challenge that the deep learning model can easily remember the samples, which induces overfitting and a bad generalization. The norm of gradient $\|\nabla_\theta \mathcal{L}(\mathcal{S}, \theta)\|$ degrades quickly during the training process. In few gradient descent steps, we will be stuck in a local minimum, where $\nabla_\theta \mathcal{L}(\mathcal{S}, \theta) = 0$.

It is meaningless to match the angle when either of the gradient norm are small. The right direction cannot help us escape the local minimum. Thus, we have to consider the magnitude of the gradient vectors in this distance function. For example, we can consider the Euclidean distance between two vectors A_i and B_i,

$$d(A, B) = \sum_{i=1}^{\texttt{out}} \left(1 - \frac{A_i \cdot B_i}{\|A_i\| \|B_i\|} + \|A_i - B_i\|\right) \tag{2.9}$$

We also try other distance functions that consider the magnitude and show the results in Section 2.1.4.3.

2.1.3.3 Where We Match: Adaptive Learning Steps

In the t-th iteration of the inner loop, θ_t is used to update $\mathcal{S}_t$, and $\mathcal{S}_{t+1}$ is used to update θ_t. Here comes the question in this sequential update. When we update $\mathcal{S}_t$, how many gradient descent steps do we perform such that $\mathcal{S}_{t+1}$ is a *good* training set for θ_t? Similarly, how many gradient descent steps do we conduct when we update θ_t such that θ_{t+1} is a *good* point for $\mathcal{S}_{t+1}$?

In the original algorithm, the authors provide their answers by setting the number of gradient descent steps $\zeta_{\mathcal{S}}$, ζ_θ empirically. Namely, we have the following update flows.

$$\mathcal{S}_t = \mathcal{S}_t^0 \to \mathcal{S}_t^1 \to \mathcal{S}_t^2 \to ... \to \mathcal{S}_t^{\zeta_{\mathcal{S}}} = \mathcal{S}_{t+1} \tag{2.10}$$

$$\theta_t = \theta_t^0 \to \theta_t^1 \to \theta_t^2 \to ... \to \theta_t^{\zeta_\theta} = \theta_{t+1} \tag{2.11}$$

We present our understanding of these two hyper-parameters. A change on $\mathcal{S}$ may induce a non-negligible update in the gradient $\nabla_\theta \mathcal{L}(\mathcal{S}, \theta)$, which is large enough for a update in the parameter. Moreover, $\mathcal{S}$ should not be updated many times at one parameter θ to avoid overfitting. Hence, the original setting of $\zeta_{\mathcal{S}} = 1$ is a good choice.

Updating θ_t is an imitation of the training process. The synthetic dataset $\mathcal{S}$ is the training dataset, while the original training dataset $\mathcal{T}$ serves as the validation dataset. In particular, after one update from θ_t^i to θ_t^{i+1}, we usually have a smaller training loss $\mathcal{L}(\mathcal{S}_{t+1}, \theta_t^i) > \mathcal{L}(\mathcal{S}_{t+1}, \theta_t^{i+1})$. However, it is unknown how the validation loss change. The relationship between $\mathcal{L}(\mathcal{T}, \theta_t^i)$ and $\mathcal{L}(\mathcal{T}, \theta_t^{i+1})$ is uncertain. We can check if there exists

overfitting after updating θ_t. The naive method of detecting overfitting is making comparison between $\mathcal{L}(\mathcal{T}, \theta_t^i)$ and $\mathcal{L}(\mathcal{T}, \theta_t^{i+1})$. We can also check the gap between two loss terms $\mathcal{L}(\mathcal{T}, \theta_t^{i+1}) - \mathcal{L}(\mathcal{S}_{t+1}, \theta_t^{i+1})$ to decide whether if overfitting happens.

Ideally, we should update network parameters θ until overfitting happens. If there is no overfitting, the $\mathcal{S}$ leads the parameter as $\mathcal{T}$ does, which means $\mathcal{S}$ is a good approximation of $\mathcal{T}$ in the perspective of gradient matching. We do not need to update $\mathcal{S}$ in this case. If overfitting happens, the $\mathcal{S}$ needs to be updated against the current parameter since the $\mathcal{S}$ has divergence from $\mathcal{T}$. Hence, it is better to use dynamic and adaptive learning steps, which help us locate where we need to update $\mathcal{S}$ and improve the algorithm efficiency.

Nevertheless, there is an overhead to detect the overfitting in real implementation. For instance, we have to compute the loss term $\mathcal{L}(\mathcal{T}, \theta_t^{i+1})$ as the extra computation. Therefore, we propose to run preliminary experiments and find when overfitting usually happens. With the preliminary results, we make a schedule for ζ_θ. In other words, let ζ_θ be a function of the index of the current inner loop t and we define this function from preliminary results to avoid the extra computation on overfitting detection. This $\zeta_\theta(t)$ could help us locate where overfitting happens approximately. Figure 2.3 shows this improvement.

2.1.3.4 Improved Gradient Matching Flow

Algorithm 1 shows our improvement on the gradient matching algorithm, with changes highlighted. We match the multi-level gradients to consider both intra-class and inter-class information without extra gradient computation, as shown in Line 8. We apply the new distance function, which considers magnitude, to avoid small gradients, which delays the overfitting. We use a dynamic number of steps in Line 11 to improve the algorithm effi-

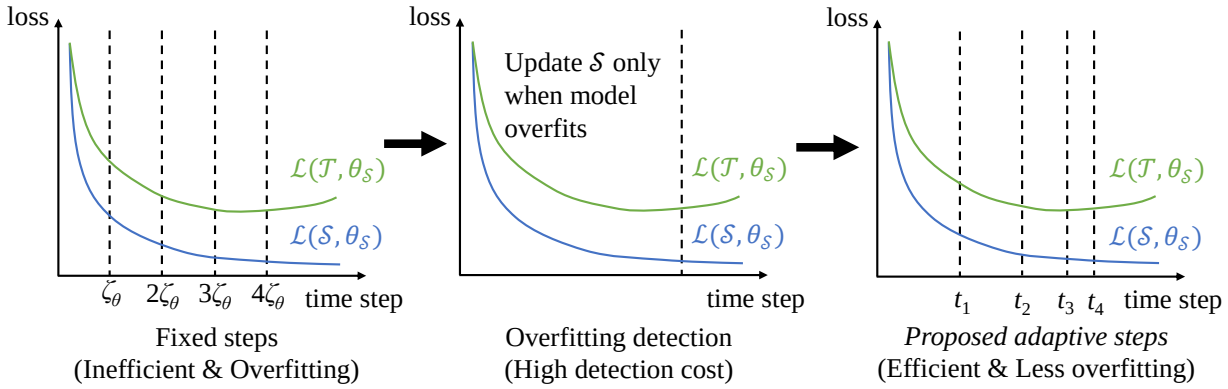


Figure 2.3: *Left.* The original method updates $\mathcal{S}$ after updating network parameter θ for a fixed number of steps ζ_θ [216]. *Middle.* Ideally, $\mathcal{S}$ should be updated right after the model overfits. *Right.* To avoid overhead on detecting overfitting, we propose to use adaptive learning steps $\zeta_\theta(t)$.

ciency. Ideally, the θ should be updated until overfitting happens where $\mathcal{S}$ diverges from $\mathcal{T}$. We use a schedule $\zeta_\theta(t)$ to help us approximate when overfitting occurs.

2.1.4 Experiments

In this section, we show an ablation study on our proposed techniques to validate their superiority to other variants and compare the test accuracy with prior arts.

2.1.4.1 Settings

We follow the same settings with the original work for all the experiments. Specifically, we use the same network architectures, datasets, and hyperparameters. The difference is the improvement highlighted in Algorithm 1. We use five image classification datasets, MNIST [40], FashionMNIST [198], SVHN [138], CIFAR-10 [98] and CIFAR-100. These datasets have a balanced class distribution. There are 100 classes in the CIFAR-100 dataset,

while other datasets have 10 classes.

We refer to the original work [216] and implementation [3] for more details regarding experimental settings. The results of the baseline method are from the original paper.

There are two phases in every single experiment. We first use our algorithm to obtain a small synthetic training set $\mathcal{S}$ with a *source* model. In the second phase, we train a *target* model with $\mathcal{S}$ from scratch and test the trained model on the original testing dataset. For every experiment, we generate 2 sets of synthetic images and train 50 target networks. The average and standard deviation of the test accuracy over these 100 evaluations are reported.

We run all our experiments on a server with Intel Core i9-7900X CPU at $3.30GHz$ and two NVIDIA Titan Xp GPUs (the CUDA version is 11.1.).

2.1.4.2 Different Gradient Matching Methods

Our first improvement is to match the multi-level gradients, which combines the intra-class gradient matching and inter-class gradient matching as discussed in Section 2.1.3.1. To demonstrate the efficacy of our proposed method, we make comparisons on the following four settings: (1) intra-class gradient matching, (2) inter-class gradient matching, (3) matching these two gradients in an interleaved way[4], and (4) multi-level gradient matching. We use the same hyperparameters in these experiments, with λ being the number of classes C, disabling the new distance function and adaptive learning steps.

Table 2.1 lists the results on four datasets. In most cases, the multi-level gradient matching achieves the best results. Focusing on either intra-class gradient matching or inter-

[3] Link to the implementation

[4] We match intra-class gradients in one iteration, and match inter-class gradients in the next iteration.

Dataset	#Image/Class	intra-class	inter-class	interleaved	multi-level
MNIST	1	**91.7±0.5**	88.8±0.7	**91.7±0.4**	90.9±0.5
	10	97.4±0.2	96.9±0.1	97.1±0.1	**97.6±0.1**
FashionMNIST	1	70.5±0.6	70.2±0.7	**70.6±0.6**	**70.6±0.7**
	10	82.3±0.4	82.4±0.3	83.1±0.3	**84.4±0.3**
SVHN	1	31.2±1.4	29.8±0.7	30.8±1.6	**32.9±1.2**
	10	**76.1±0.6**	72.7±1.0	75.4±0.7	75.5±0.7
CIFAR-10	1	28.3±0.5	**29.7±0.7**	28.6±0.7	**29.7±0.7**
	10	44.9±0.5	46.7±0.5	45.9±0.6	**48.6±0.5**

Table 2.1: Test accuracy (%) with matching different gradients. New distance function and adaptive learning steps are disabled.

class gradient matching misses the other information. Compared with minimizing the two matching losses in an interleaved way, the multi-level gradient matching is much more stable.

2.1.4.3 Different Distance Functions

Our second improvement is to use a new distance function. Instead of only focusing on the angle between gradients, we also match the magnitude. Here, we make comparisons on the following distance functions, with multi-level gradients enabled. (1) Negative cosine similarity $d_1(a, b) = 1 - (a \cdot b)/(\|a\|\|b\|)$, (2) Euclidean distance $d_2(a, b) = \|a - b\|$, (3) sum of the squared error $d_3(a, b) = \|a - b\|^2$, (4) mean squared error $d_4(a, b) = d_3(a, b)/len(a)$, Note that the d_1 focuses on the angle only, while the other functions consider angle and magnitude simultaneously.

We take the SVHN dataset with 1 image per class as an example. Table 2.2 lists the test accuracy when using these distance functions to generate synthetic sets. We directly assign the same weight to these distance functions except that we set the weight of 100 for d_4.

Distance Function	Test Accuracy
$d_1(a,b) = 1 - (a \cdot b)/(\|a\|\|b\|)$	32.9±1.2
$d_2(a,b) = \|a-b\|$	23.6±2.1
$d_3(a,b) = \|a-b\|^2$	24.3±1.8
$100d_4(a,b) = 100d_3(a,b)/len(a)$	23.5±1.4
$d_1 + d_2$	**34.5±1.9**
$d_1 + d_3$	34.1±2.0
$d_1 + 100d_4$	34.0±1.4

Table 2.2: Test accuracy (%) with different distance functions.

We find that when using only one of these distance functions, the test accuracy with d_1 is the highest. Our explanation is that the angle of the gradient is much more important than the magnitude when (stochastic) gradient descent and its variants are used. However, when we combine d_1 and other magnitude-related distance functions, we get improvement compared with pure d_1. Namely, we concentrate on the gradient direction while considering the magnitude to avoid being stuck in traps where the gradient norm is small.

2.1.4.4 Adaptive Learning Steps

Ideally, we should update $\mathcal{S}$ when it is no longer a good approximation of $\mathcal{T}$ in terms of gradient matching. Thus, a criterion to detect overfitting is needed. We try the naive overfitting criterion $\mathcal{L}(\mathcal{T}, \theta_t^i) < \mathcal{L}(\mathcal{T}, \theta_t^{i+1})$. In other words, if the validation loss increases, we will stop the parameter update and proceed to update $\mathcal{S}$. With this setting, we have improved the test accuracy from 44.9% to 45.7% for 10 images per class of CIFAR-10. However, we notice that this improvement is at the cost of overfitting detection, which is nontrivial in real implementation. Therefore, we define a pre-defined schedule $\zeta_\theta(t)$ by observing when

overfitting happens in the CIFAR-10 experiment above.

$$\zeta_\theta(t) = \begin{cases} 50 - 10t, & t < 4 \\ 10, & 4 \leq t < 10 \\ 5, & t \geq 10 \end{cases} \tag{2.12}$$

The reason $\zeta_\theta(t)$ is non-increasing is that we may encounter overfitting issues more frequently as training proceeds. Hence, we need to update $\mathcal{S}$ at shorter intervals.

With this schedule, we can first proceed to where overfitting happens and then stay in this area. Another advantage of this schedule is that it reduces the number of model parameter updates. In the baseline method, the authors set $T = 1, 10, 50, \zeta_\theta = 1, 50, 10$ for synthesizing $1, 10, 50$ images per class. Taking $T = 10$ as an example, the original algorithm updates θ 450 times in one iteration of the outer loop, while we only update it 190 times.

2.1.4.5 Comparison with Prior Work

In Table 2.3, we perform an ablation study on our methods with four settings to demonstrate the effectiveness of our proposed enhancement. We name the four settings as *Ours-M*, *Ours-MD*, *Ours-MDO*, and *Ours-MDA*, where M, D, O, A stand for **m**ulti-level gradient matching, new **d**istance function, updating θ until **o**verfitting, and **a**daptive steps, respectively. We also add two coreset selection methods for comparison. *Random* means that samples are randomly selected as the coreset. *Herding* [9, 34] selects samples whose center is close to the distribution center. For a fair comparison, we evaluate our method on the same ConvNet model [55] as used in the original work [216]. Both the source network and the target network are the ConvNet model.

For our method, we use the settings mentioned above. We match the multi-level gradients as shown in Equation equation 2.6 with $\lambda = C$, the number of classes. We use

Dataset	IPC	Random	Herding	DC Baseline	Ours-M	Ours-MD	Ours-MDO	Ours-MDA	Whole Training Set
MNIST	1	64.9±3.5	89.2±1.6	91.7±0.5	90.9±0.5	**91.9±0.4**	-	-	99.6±0.0
	10	95.1±0.9	93.7±0.3	97.4±0.2	97.6±0.1	**97.9±0.1**	**97.9±0.1**	**97.9±0.2**	
	50	97.9±0.2	94.8±0.2	**98.8±0.2**	98.0±0.1	98.6±0.1	98.6±0.1	98.5±0.1	
FashionMNIST	1	51.4±3.8	67.0±1.9	70.5±0.6	70.6±0.7	**71.4±0.6**	-	-	93.5±0.1
	10	73.8±0.7	71.1±0.7	82.3±0.4	84.4±0.3	**85.4±0.3**	84.6±0.3	84.2±0.3	
	50	82.5±0.7	71.9±0.8	83.6±0.4	87.8±0.2	87.4±0.2	**87.9±0.2**	**87.9±0.2**	
SVHN	1	14.6±1.6	20.9±1.3	31.2±1.4	32.9±1.2	**34.5±1.9**	-	-	95.4±0.1
	10	35.1±4.1	50.5±3.3	76.1±0.6	75.5±0.7	75.9±0.7	**76.2±0.7**	75.9±0.7	
	50	70.9±0.9	72.6±0.8	82.3±0.3	82.2±0.2	82.9±0.2	**83.8±0.3**	83.2±0.3	
CIFAR-10	1	14.4±2.0	21.5±1.2	28.3±0.5	29.5±0.7	**30.0±0.6**	-	-	84.8±0.1
	10	26.0±1.2	31.6±0.7	44.9±0.5	48.6±0.5	49.5±0.5	49.9±0.6	**50.2±0.6**	
	50	43.4±1.0	40.4±0.6	53.9±0.5	58.5±0.5	58.6±0.4	**60.0±0.4**	58.3±0.5	
CIFAR-100	1	4.2±0.3	8.4±0.3	**12.8±0.3**	12.4±0.3	12.7±0.4	-	-	56.2±0.3
	10	14.6±0.5	17.3±0.3	25.2±0.3	30.8±0.3	28.0±0.4	29.5±0.3	**31.1±0.3**	

Table 2.3: Ablation study in terms of the test accuracy (%). IPC is the number of image per class in $\mathcal{S}$. *Random* means that samples are randomly selected as the coreset. *Herding* selects samples whose center is close to the distribution center. *DC baseline* refers to the original work on dataset condensation. M, D, O, A represent multi-level gradient matching, new distance function, updating θ until overfitting, and adaptive steps, respectively. O and A are not applicable when IPC is 1.

the distance in Equation equation 2.9, the overfitting criterion $\mathcal{L}(\mathcal{T}, \theta_t^i) < \mathcal{L}(\mathcal{T}, \theta_t^{i+1})$, and the adaptive learning step in Equation equation 2.12. [5] We use the same settings for all the benchmarks without further tuning [6]. It is expected that we can achieve better results with better hyperparameters tuned for each benchmark. For instance, we can tune the distance function and the learning steps $\zeta_\theta(t)$.

Since the algorithm only runs a single inner loop, i.e., $T = 1$, when the condensed dataset contains one image per class, the method of adaptive step has no impact in this setting. In terms of test accuracy, our proposed multi-level gradient matching and angle-magnitude distance function outperform the baseline gradient matching method [216] in

[5] Equation equation 2.12 is from the observation on a CIFAR-10 experiment and we generalize the setting to other benchmarks.

[6] An exception is that we use $d = d_1 + 0.1d_2 = 1 - (a \cdot b)/(\|a\|\|b\|) + 0.1\|a - b\|$ for 50 images per class with CIFAR-10.

most benchmarks. Our proposed adaptive learning step technique is a good approximation of overfitting detector since the results of *Ours-MDO* and *Ours-MDA* are similar. With *Ours-MDA*, we cut down unnecessary steps in the later optimization stage, leading to higher learning efficiency while maintaining our advantages in test accuracy.

Regarding the algorithm efficiency, the usage of multi-level gradients and new distance functions introduces less than 1% extra computation time. The adaptive learning step can reduce the computation time by $25\% \sim 30\%$ for the experiments with 10 images per class.

Although training on condensed datasets has a performance degradation, we argue that condensed datasets are usually used for quick training and training with limited resources. If final training performance is the only objective, we have to conduct training on the whole dataset. For reference, the baseline method achieves $64\%, 67\%, 71\%, 77\%, 83\%$ relative accuracy (the ratio compared to training on full dataset) with 50, 100, 200, 500, 1000 images per class on CIFAR-10 dataset. We achieve $71\%, 74\%, 78\%, 84\%, 89\%$ relative accuracy in these settings.

One of the limitations is that we do not conduct experiments on large datasets for two reasons. First, the exploration is quite computationally intensive so that we cannot handle these experiments. Second, most of the previous work focuses on these benchmarks and we directly follow the same settings. Moreover, we conduct experiments on tiny ImageNet (200 classes, image size 64×64, 1 image per class) with the same experiment settings as CIFAR-10/100. The test accuracy (%) of the baseline is 4.65 ± 0.20, while ours is 4.93 ± 0.23.

source\target	MLP	ConvNet	LeNet	AlexNet	VGG	ResNet
MLP	67.3±1.1	71.2±1.6	70.5±8.5	45.1±12.4	46.9±4.2	83.2±2.1
ConvNet	72.7±1.6	91.9±0.4	84.1±2.2	84.1±2.4	84.3±1.5	90.4±0.4
LeNet	62.8±1.6	87.2±0.7	81.8±1.9	80.8±3.6	75.1±2.5	88.1±1.1
AlexNet	61.3±1.6	87.6±0.8	81.4±2.5	81.2±3.3	77.7±2.1	87.9±1.0
VGG	63.8±2.3	89.3±0.7	82.4±2.4	82.5±2.8	81.1±2.2	89.4±0.8
ResNet	62.2±1.8	82.3±2.3	80.4±3.2	80.1±2.5	75.7±3.2	87.4±1.0

Table 2.4: Cross-generalization test accuracy (%). We condense the training set on one source network and test the resultant synthetic set on other target networks.

2.1.4.6 Cross-Architecture Generalization

Following the same setting as the original work, we present our experiments on the cross-architecture generalization with *Ours-MD*. We use the network architecture of multi-layer perceptron (MLP), ConvNet [55], LeNet [105], AlexNet [99], VGG-11 [167], and ResNet-18 [63].

Table 2.4 shows the results when the source and target networks are different for 1 image per class of the MNIST dataset condensation. We obtain a similar result to the original work [216]. The datasets learned from the convolutional neural architectures (ConvNet, LeNet, AlexNet, VGG, and ResNet) generalize to the other convolutional networks. Since the hyperparameters are searched based on the ConvNet, all the target networks achieve the highest test accuracy when trained from the dataset learned from ConvNet. Moreover, the learned dataset can be used to validate the neural architectures. For instance, we find that whatever the source network is, ResNet achieves one of the highest test accuracies among all target networks.

2.1.5 Discussion

In this section, we present our qualitative analysis of the gradient matching method and our improvement. When training a neural network using gradient descent or its variants, there is a reachable region in the parameter space. We explore and enlarge this space from the initialization point throughout the training process. Finally, we will select the best parameters in this region. The gradient matching algorithm matches the first-order loss landscape of this reachable region, ignoring the unachievable parameter space.

We hypothesize that the gradient matching algorithm is actually the following optimization problem.

$$\max_{\mathcal{S}} \min_{\theta,\theta_0} \quad \|\theta - \theta_0\| \tag{2.13}$$

$$\text{s.t.} \quad \|\nabla_\theta \mathcal{L}(\mathcal{S}, \theta)\| < \epsilon \tag{2.14}$$

$$\theta_0 \sim P_{\theta_0} \tag{2.15}$$

Here ϵ is a threshold such that the parameter satisfying $\|\nabla_\theta \mathcal{L}(\mathcal{S}, \theta)\| < \epsilon$ cannot escape this point with gradient descent method. We maximize the distance between any pairs of the local minimum (Equation 2.14) and the initialization point (Equation 2.15). Using gradient matching, we attempt to eliminate these reachable local minimums by updating the synthetic set $\mathcal{S}$. Thus, we can proceed further and explore this reachable region when training with $\mathcal{S}$. Each iteration of the outer loop of the Algorithm 1 is a depth-first search of this region. We update $\mathcal{S}$ to escape from the traps in this region and update θ to explore and enlarge the reachable region. We try different initialization to mimic the minimization in Equation equation 2.13.

Our three improvement can be interpreted with this hypothesis. The multi-level gradients consider both intra-class and inter-class information. Matching inter-class gradients can help us synthesize images concentrating on the reachable region since we use inter-class mini-batches when using the resultant $\mathcal{S}$. The intra-class gradient matching can accelerate the convergence. With the distance function considering the magnitude, we can mitigate the internal local minimum and enlarge the reachable area. The adaptive learning step can help us proceed to the internal local minimum or the boundary of this region to do updates efficiently. Above all, our proposed techniques are used to remove the local minimum in the reachable region of the parameter space.

2.1.6 Summary

We present our analysis of the gradient matching method for the dataset condensation problem. Based on our analysis, we further extend the original algorithm. We provide our answers to the question of *what, how, and where we match in this gradient matching algorithm.* We match the multi-level gradients to involve both intra-class and inter-class gradient information. A new distance function is proposed to mitigate the overfitting issue. We use adaptive learning steps to improve algorithm efficiency. The effectiveness and efficiency of our proposed improvements are shown in the experiments.

We notice that the dataset condensation and distillation are getting more and more attention. There are standard benchmarks [39] and comprehensive surveys [54], which present several future directions, such as improving the computational efficiency and extending to large datasets. Other than that, we are interested in extending the dataset condensation into semi-supervised learning. It is worth exploring if we can compress the huge corpus to

train large language models more efficiently.

2.2 NormSoftmax: Normalizing the Input of Softmax to Accelerate and Stabilize Training

Softmax is a critical and widely used function in machine learning algorithms, which takes a vector as input and generate a standard simplex. It is usually used to generate a categorical probability distribution. The most notable applications of softmax are cross-entropy loss function for classification tasks and attention map generation in dot product attention operations. By importing the temperature in softmax, we can control the information entropy and sharpness of its output.

However, gradient-based optimization of softmax-based models often suffers from slow and unstable convergence and is sensitive to optimization hyperparameters. Transformer-based models [183] are known to be hard to optimize. A lot of efforts have been devoted to solving this optimization difficulty [117]. For instance, Bolya *et al*. [15] reports that softmax attention may crash with too many heads and proposes new attention functions. Chen *et al*. [33] show that the Vision Transformer's [47] loss landscape is very sharp, and it requires advanced optimizers to facilitate its training [53]. Huang *et al*. [74] propose a better initialization to improve the Transformer optimization. Xiong *et al*. [201] show that the location of layer normalization (LN) has a remarkable impact on the gradients and claim that the Pre-LN Transformer has better training stability.

Among comprehensive reasons for the optimization difficulty of Transformers, cascaded softmax functions are one of them that leads to the training instability. However, limited prior work has discussed the impacts of softmax on optimization. Based on our

experimentation, we find that the training difficulty can be attributed to the rapid change in the variance of the softmax inputs and the information entropy of its outputs. In dot-product attention, where the softmax is used to generate weight distribution for key-value pairs, we observe significant statistical fluctuation in softmax inputs. The rapid and extensive variance change in the initial learning stage can lead to unstable training. Moreover, for the softmax used in cross-entropy loss for classification problems, the input of the softmax usually has a lower variance at the initial training stage since the model has less knowledge of the problem [188]. The model is likely to stay in the low-confidence zone, implying that it is challenging to train [143]. We need a specially designed mechanism to push the model out of this low-confidence zone for stable and fast learning.

2.2.1 Overview

In the two cases above, the significant change in the softmax input variance is one of the reasons for optimization difficulty. In this section, we propose NormSoftmax to stabilize and accelerate training by simply re-scaling the softmax inputs, especially in the early-stage optimization.

With NormSoftmax, we dynamically calculate vector-specific factors to scale the inputs before being fed to the standard softmax. Specifically, when the input variance is too small, Softmax will generate small gradients that hinder the learning process. In contrast, our proposed NormSoftmax can help re-scale the input distribution such that the information entropy of the output becomes stable without fluctuation during the training process, which boosts and stabilizes the early-stage training.

NormSoftmax shares similar properties with the existing normalization techniques in

machine learning. We summarize its advantages below.

- NormSoftmax can re-scale gradients to stabilize the training process, making the training robust to model architectures and optimization recipes (such as optimizers and weight decay schedules).
- NormSoftmax can accelerate the early training stage without hurting the model representability.
- NormSoftmax is an easy-to-use and low-cost module to replace standard softmax. The induced computation and memory cost overhead is negligible.
- NormSoftmax has a regularization effect since the re-scaling can slightly restrict the representation space of the input vectors.

In this section, we focus on two applications of the softmax functions: (1) the activation function in dot-product attention, and (2) cross-entropy loss of the classification problem. ViT-B with our NormSoftmax shows significantly higher robustness to different head settings, showing an average of $+\mathbf{4.63\%}$ higher test accuracy on CIFAR-10 than its softmax-based counterpart. When training for 100 epochs on ImageNet-1K, ViT with our NormSoftmax can achieve $+\mathbf{0.91\%}$ higher test accuracy over its softmax baseline.

2.2.2 Background

We briefly introduce the softmax function and normalization in machine learning. Then we discuss the two cases we focus on in this section: softmax in dot product attention and cross entropy loss. We use $\mu(\boldsymbol{a}), \sigma(\boldsymbol{a})$ to represent the mean and standard deviation (square root of the variance) of a vector $\boldsymbol{a}$.

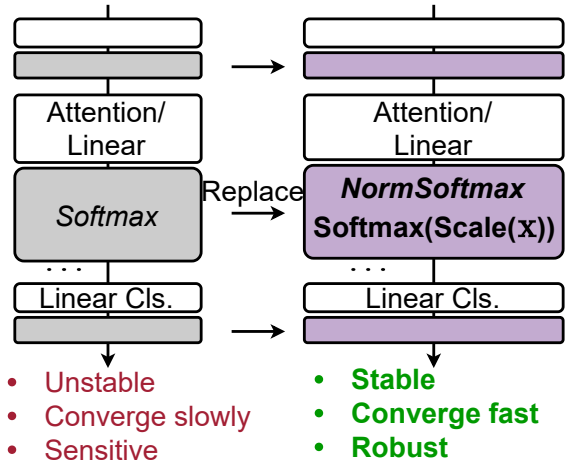


Figure 2.4: The standard softmax and proposed NormSoftmax.

2.2.2.1 Softmax

The standard softmax function [20] $\boldsymbol{z} = \text{softmax}(\boldsymbol{x})$, where $\boldsymbol{x}, \boldsymbol{z} \in \mathbb{R}^n$ is defined by Equation 2.16.

$$z_i = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}, \text{for } i = 1, 2, ..., n \tag{2.16}$$

The output of softmax can be seen as a categorical probability distribution since $0 < z_i < 1$ and $\sum_i z_i = 1$. Instead of e, we can also use a different base in softmax. A temperature parameter $T > 0$ is imported to adjust the base.

$$\text{softmax}_T(\boldsymbol{x}) = \text{softmax}\left(\frac{\boldsymbol{x}}{T}\right) \tag{2.17}$$

Given the same input vector $\boldsymbol{x}$, the higher temperature smooths the difference of the input vector and generates a probability distribution with high information entropy $H(\boldsymbol{z}) = -\sum_i z_i \log(z_i)$. On the contrary, the lower temperature sharpens the output distribution with low entropy. Agarwala *et al.* [2] claim that the temperature has a crucial impact on the initial learning process.

2.2.2.2 Normalization

Normalization layers [5, 76, 182, 197] usually normalize the input tensor such that the result has a zero-mean and unit-variance along specific dimensions. We can optionally scale and shift the normalized tensor further. Normalization can accelerate and stabilize the optimization by smoothing the loss landscape [13, 162, 202]. Hence, normalization allows for a larger learning rate and increases the robustness against hyperparameters. Also, normalization helps generalization since the sharpness of the loss surface is decreased effectively [124]. However, we find that normalization is usually used in the intermediate linear transformation layers, and it is rarely applied to the input of softmax functions.

2.2.2.3 Dot Product Attention

The scaled dot product attention [6, 183] is defined by the following equation, where $\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}$ are query, key, and value matrices and d is the dimension of the key vector.

$$\text{Attention}(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}) = \text{softmax}\left(\frac{\boldsymbol{Q}\boldsymbol{K}^T}{\sqrt{d}}\right)\boldsymbol{V} \tag{2.18}$$

For every query vector, the softmax function calculates the weight for all key-value pairs. The scaling factor of $d^{-1/2}$ is proposed to attempt the normalize the dot product $\boldsymbol{q}^T\boldsymbol{k}$, whose variance is d if the components of $\boldsymbol{q}, \boldsymbol{k}$ are independent random variables with a variance of 0. [7] The scaling factor is applied to address the issue that the variance of dot product $\boldsymbol{q}^T\boldsymbol{k}$ will likely increase as the length of the vector d increases. The large variance makes the gradient of softmax extremely small, thus making the attention-based model hard to train.

[7] Please refer to Footnote 4 of the original paper [183].

Based on self-attention, Transformer has achieved great success in many areas, especially natural language processing [45] and computer vision [91].

2.2.2.4 Cross Entropy Loss

Another important application of softmax is in classification problems, where minimizing the cross-entropy loss is equivalent to maximizing the likelihood. The cross-entropy function takes the estimated probability distribution $\boldsymbol{q} = \text{softmax}(\boldsymbol{x})$ and the true probability distribution $\boldsymbol{p}$ as input and computes the result by $H(\boldsymbol{p}, \boldsymbol{q}) = -\sum_i p_i \log q_i$. $\boldsymbol{x}$ is the predicted logits, usually generated by a classification model. $\boldsymbol{x}$ can be any vector in $\mathbb{R}^K$ without restrictions, where K is the number of classes.

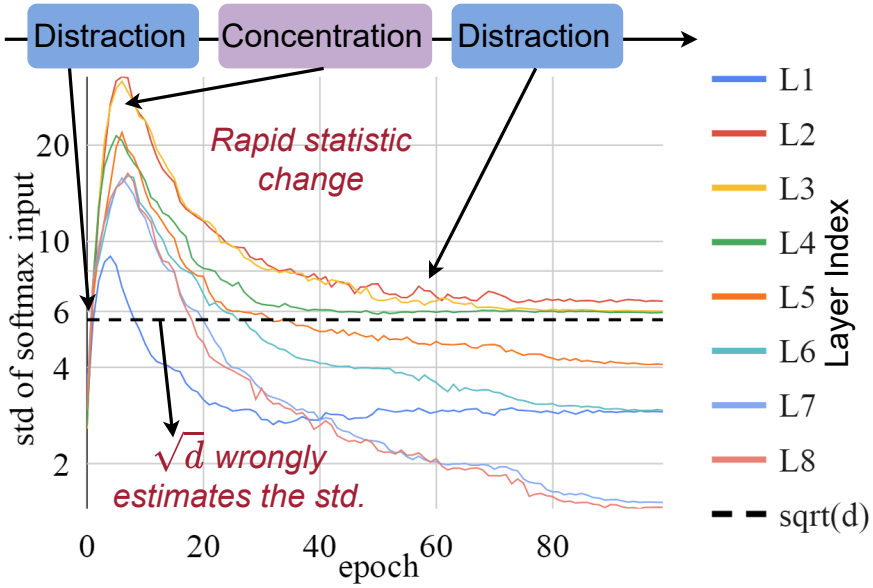
2.2.3 Method

In this section, we first analyze the behavior of the softmax input during the training process. Further, we define NormSoftmax and discuss its advantage in the two cases.

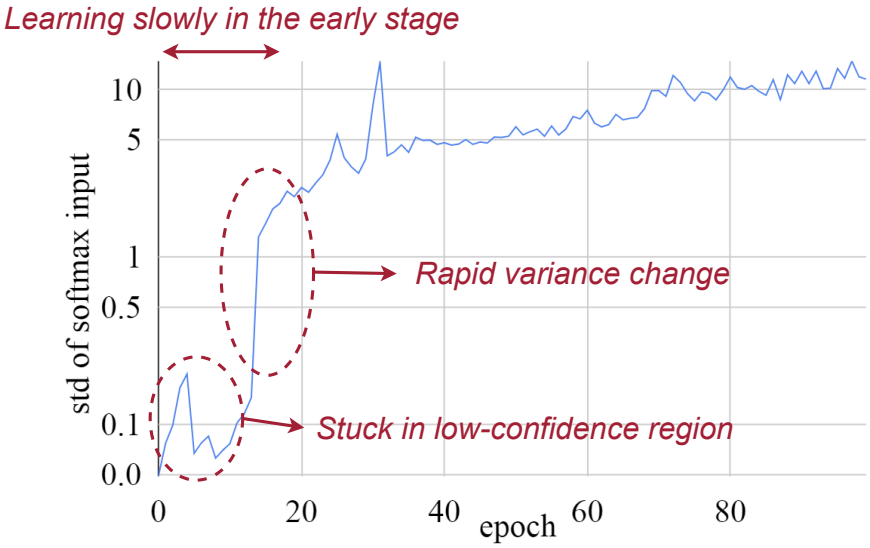
2.2.3.1 Softmax in Different Training Stages

We split the whole training process into three stages.

- **Initial stage.** Starting from scratch, we usually need a careful design for initialization and hyperparameters. The training may fail due to exploding or vanishing gradients. For example, we usually apply learning rate warmup during this stage.
- **Intermediate stage.** Once the parameters are sufficiently warmed up, it is relatively easy and stable to explore the solution space with a higher learning rate.



a Softmax in Attention of an 8-layer ViT



b Softmax for the cross entropy loss in ResNet1202

Figure 2.5: The standard deviation of softmax input has significant change during the training process. Note that the vertical axes are in the logarithmic scale.

- **Final stage.** The model attempts to converge to a local minimum with a lower learning rate.

The initial stage is the most unstable one among these three stages. If we do not encounter severe issues in the initial stage, it is likely that we can proceed to the final results.

Figure 2.5 illustrates two cases where we observe a significant change in the softmax input variance throughout the training process. We discuss the behavior of the softmax function in these two cases.

Softmax in dot product attention. The input variance of softmax is related to the attended region of this attention layer. With a high variance on the input, the output of softmax has a low information entropy, meaning that only few keys are attended. On the contrary, the smaller input variance implies that much more keys are attended. Figure 2.5a demonstrates that the standard deviation of softmax input (before scaled with $\sqrt{d}$) increases rapidly at the initial stage and then decreases gradually during the intermediate and final stages. In the beginning, the vision Transformer [47] attempts to attend to almost all the keys since the initialized model has not learned how to extract features and is still in the exploration stage. During training, different keys are learned at different paces. Keys with smaller semantic distances from nearby queries are much easier to learn. This imbalanced learning pace among keys drives the model to shrink its receptive field and focus on a small region. That is why the variance of softmax input increases rapidly at the initial stage. Afterwards, as training proceeds, Transformer will explore the query-key pairs with longer semantic distance, implied by a gradual reduction in the variance of softmax input. We name the effect "distraction-concentration-distraction".

Further, different layers have different input variances. Except for the first layer (L1), the lower layer has a high variance than the deeper layers. In the early layers, only a small region is attended to. As depth grows, the model attends to a larger region, which is similar to the trend of receptive fields in convolutional neural networks. We also draw the line of $\sqrt{d}$ as a reference in Figure 2.5a, which is the scaling factor used to normalize the softmax input. Since the *input variance has a significant change across different layers and training steps, this constant scaling factor of* $\sqrt{d}$ *might not be the most suitable value* to normalize the softmax input [107].

Softmax in the cross-entropy loss function. For classification model training, the softmax may only appear in the cross-entropy loss function, e.g., ResNet. However, the gradients through softmax can have a large impact on the model training. To verify this claim, we train a deep ResNet1202 [63] on the CIFAR10 dataset [98] and plot the standard deviation of the softmax input in Figure 2.5b. At the initial stage, the standard deviation is relatively low since the model is less confident, and the predicted distribution is similar to a uniform distribution. There is a leap at the 15-th epoch, where the standard deviation increases from 0.14 to 1.31. Afterward, the standard deviation of the softmax input gradually increases, and the information entropy of the softmax output becomes smaller since the model becomes more confident in predictions as training continues. Hence, we conclude that the variance of softmax input experiences rapid and huge change during training, especially in the initial stage, which explains why training from scratch is difficult.

Moreover, the observed trend in Figure 2.5 is averaged on all the data points. Different data points or training examples are in different learning stages. For instance, in the image classification problem, a clear image can easily escape the less-confident zone quickly and

stably, while a vague image may be stuck in this zone where the softmax output has a high information entropy. Therefore, this training difficulty needs to be tackled in a data-specific fashion.

2.2.3.2 The Proposed NormSoftmax

We propose NormSoftmax as a substitution for softmax, as defined below,

$$\text{NormSoftmax}(\boldsymbol{x}, \gamma) = \text{softmax}\left(\frac{\boldsymbol{x} - \mu(\boldsymbol{x})\mathbf{1}}{\min(\sigma(\boldsymbol{x}), \gamma)}\right) \tag{2.19}$$

$$= \text{softmax}\left(\frac{\boldsymbol{x}}{\min(\sigma(\boldsymbol{x}), \gamma)}\right) \tag{2.20}$$

$$= \text{softmax}\left(\frac{\boldsymbol{x}/\gamma}{\min(\sigma(\boldsymbol{x}/\gamma), 1)}\right) \tag{2.21}$$

where $\gamma > 0$ is a pre-defined scalar. Namely, we define the temperature $T = \min(\sigma(\boldsymbol{x}), \gamma)$ in NormSoftmax. Since softmax is invariant under translation by the same value, Equations 2.19 and 2.20 are equivalent. We do not need to shift the input vector to zero-mean. If the standard derivation is smaller than the threshold $\sigma(\boldsymbol{x}) \leq \gamma$, NormSoftmax will normalize the input vector to obtain a unit-variance vector before applying the standard softmax function. If $\sigma(\boldsymbol{x}) > \gamma$, we use the temperature $T = \gamma$. If $\gamma = +\infty$, then we will always normalize the input vector. The temperature is dynamically calculated *per vector*. For a batch of vectors, each one has its individual temperature.

Lemma 2.2.1 *Given* $\boldsymbol{y} = \frac{\boldsymbol{x}}{\sigma(\boldsymbol{x})}, \boldsymbol{z} = softmax(\boldsymbol{y}), \frac{\partial l}{\partial \boldsymbol{z}} \in \mathbb{R}^n$*, we have*

$$\mu\left(\frac{\partial l}{\partial \boldsymbol{x}}\right) = \mu\left(\frac{\partial l}{\partial \boldsymbol{y}}\right) = 0, \tag{2.22}$$

$$\sigma\left(\frac{\partial l}{\partial \boldsymbol{x}}\right) \leq \frac{\sigma\left(\frac{\partial l}{\partial \boldsymbol{y}}\right)}{\sigma(\boldsymbol{x})}, \tag{2.23}$$

$$\left\|\frac{\partial l}{\partial \boldsymbol{x}}\right\|_2 \leq \frac{\left\|\frac{\partial l}{\partial \boldsymbol{y}}\right\|_2}{\sigma(\boldsymbol{x})} \tag{2.24}$$

Lemma 2.2.2 *Given* $\boldsymbol{x}_2 = k\boldsymbol{x}_1, \boldsymbol{z}_1 = softmax(\boldsymbol{x}_1/\sigma(\boldsymbol{x}_1)), \boldsymbol{z}_2 = softmax(\boldsymbol{x}_2/\sigma(\boldsymbol{x}_2)), l_1 = f(\boldsymbol{z}_1), l_2 = f(\boldsymbol{z}_2)$*, we have* $\boldsymbol{z}_1 = \boldsymbol{z}_2, l_1 = l_2, \frac{\partial l_1}{\partial \boldsymbol{z}_1} = \frac{\partial l_2}{\partial \boldsymbol{z}_2}, \frac{\partial l_2}{\partial \boldsymbol{x}_2} = \frac{1}{k}\frac{\partial l_1}{\partial \boldsymbol{x}_1}$.

Similar to the theorem in [202], we refer to gradient re-centering and re-scaling as gradient normalization. Similar to Theorem 4.1 in [162], we demonstrate that the normalization improves the Lipschitz continuity, indicated by the gradient magnitudes. Specifically, $\left\|\frac{\partial l}{\partial \boldsymbol{x}}\right\|_2$ and $\left\|\frac{\partial l}{\partial \boldsymbol{y}}\right\|_2$ can be treated as the continuity of the loss function. Given two input vectors $\boldsymbol{x}_1, \boldsymbol{x}_2$ and $\sigma(\boldsymbol{x}_1) < \sigma(\boldsymbol{x}_2) < \gamma$, NormSoftmax apply a different temperature on them, paying much attention to the low-variance vector $\boldsymbol{x}_1$. In Lemma 2.2.2, if $\sigma(\boldsymbol{x}_1) = 1$, then $k = \sigma(\boldsymbol{x}_2)$, it is clear that the gradients are rescaled by its variance $\frac{\partial l}{\partial \boldsymbol{x}_2} = \frac{1}{\sigma(\boldsymbol{x}_2)}\frac{\partial l}{\partial \boldsymbol{x}_1}$.

Logit Normalization (LogitNorm) [96, 188] uses the ℓ_2 norm to normalize the input vector, as shown in the equation below, where τ is the temperature parameter modulating the magnitude of the logits.

$$\text{LogitNorm}(\boldsymbol{x}, \tau) = \text{softmax}\left(\frac{\boldsymbol{x}}{\tau\|\boldsymbol{x}\|_2}\right) \tag{2.25}$$

LogitNorm is proposed to mitigate overconfidence when cross entropy loss is used. It is questionable that LogitNorm does not shift the input vector to zero-mean, since the mean

has an impact on the ℓ_2 norm. Unlike the standard softmax function, LogitNorm is not invariant under translation $\text{LogitNorm}(\boldsymbol{x}, \tau) \neq \text{LogitNorm}(\boldsymbol{x} + c\mathbf{1}, \tau)$, where c is a constant.

We demonstrate the relationship between LogitNorm and NormSoftmax in the following equation.

$$\text{NormSoftmax}(\boldsymbol{x}, \gamma = +\infty) = \text{LogitNorm}(\boldsymbol{x} - \mu(\boldsymbol{x})\mathbf{1}, \tau = n^{-1/2}) \tag{2.26}$$

NormSoftmax first shifts the input vector to zero-mean and normalizes the shifted vector by its ℓ_2 norm. Hence, NormSoftmax keeps the invariance under translation and can be reduced to LogitNorm with input shifting.

2.2.3.3 Effects of NormSoftmax in Three Stages

NormSoftmax can accelerate and stabilize training in the initial stage. With NormSoftmax, the standard deviation of the softmax input is at least 1 since $\sigma\left(\frac{\boldsymbol{x}}{\min(\sigma(\boldsymbol{x}),\gamma)}\right) \geq 1$. If the softmax input has low variance, which is common in the initial stage for both two applications, we use a self-adapted low temperature to magnify the slight difference. The normalization can help the model escape the zone with high information entropy quickly and stably since we normalize the gradients as demonstrated in Theorem 2.2.1. Specifically, for attention layers, NormSoftmax can accelerate the transition from distraction to concentration.

NormSoftmax can regularize training in the intermediate and final stages. For cross entropy loss, NormSoftmax increases the temperature of the softmax in the training process since the variance of the softmax input will increase gradually. The high temperature can regularize the training. For attention layers, NormSoftmax regularizes the softmax input,

thus restricting the attentive areas. NormSoftmax encourages the attention to have a slight change on the attended regions, which can be treated as an inductive bias we add.

However, we find that by simply normalizing inputs to unit-variance vectors, i.e., $\boldsymbol{x}/\sigma(\boldsymbol{x})$, the training process can be impeded due to overly restricted representation space. Our variance clipping technique can effectively solve this issue with a pre-defined threshold γ.

2.2.4 Experiments

Detailed experiment settings can be found in Appendix. Since we primarily use the cosine annealing learning rate scheduler, we train from scratch when the total number of epochs is different. Namely, the learning rate schedule is updated according to the number of epochs.

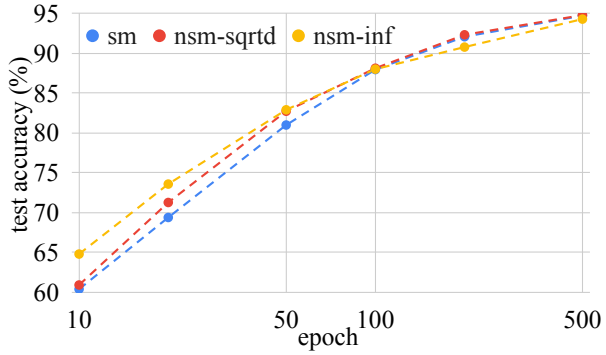
2.2.4.1 Dot-Product Attention

baseline	f_1	$f_2, \gamma = \sqrt{d}$	nsm-1	nsm-$\sqrt{d}/2$	nsm-$\sqrt{d}$	nsm-$2\sqrt{d}$	nsm-∞	logitnorm
87.96	87.97	86.99	81.04	87.16	88.13	88.21	88.01	87.72

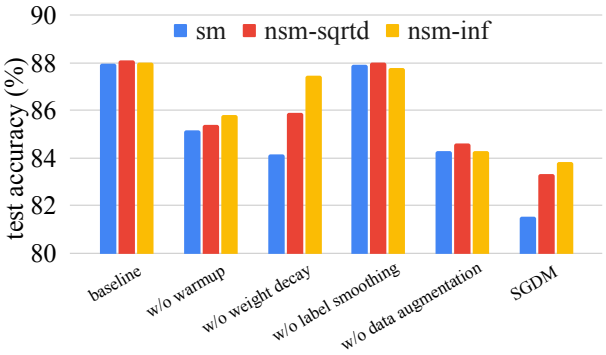
Table 2.5: Train a ViT on CIFAR10 with different NormSoftmax variants.

Settings. We train the vision transformer (ViT) on the CIFAR10 dataset from scratch with AdamW [121] optimizer for 100 epochs (50,000 iterations with 100 mini-batch size). The resolution of an input image is $3 \times 32 \times 32$, and the patch size is 4. The hidden size, MLP size, number of heads, and the dimension of heads are 256, 1024, 8, and 32, respectively. We discard the original scaling factor $\sqrt{d}$ and replace the standard softmax with NormSoftmax, setting $\gamma = \sqrt{d}$ or $\gamma = +\infty$. The learning rate is linearly increased

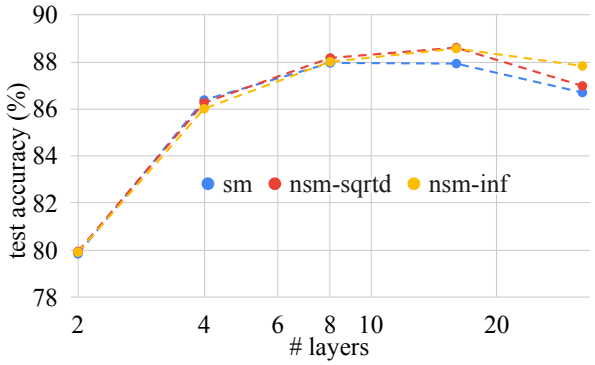
with 5 warmup epochs and then decays with the cosine annealing scheduler [120]. Following the original ViT paper [47], the learning rate is $1e-3$, and a strong weight decay $1e-1$ is applied. We also enable label smoothing [175] and strong data augmentation (random erasing [218], mixup [211], cutmix [206], and TrivialAugment [135]). Figure 2.6 shows the detailed results.



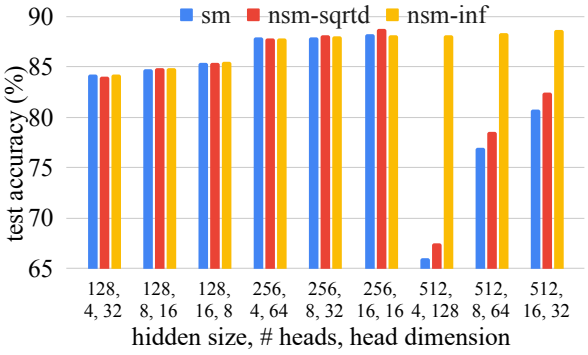
a Results with different training epochs



b Results with different training recipes



c Results on ViT with different depths



d ViT with different dimensions and heads

Figure 2.6: The test accuracy of a ViT on CIFAR10 dataset with different settings. `sm` and `nsm` are short for softmax and NormSoftmax. `sqrtd` and `inf` are the value of γ in NormSoftmax.

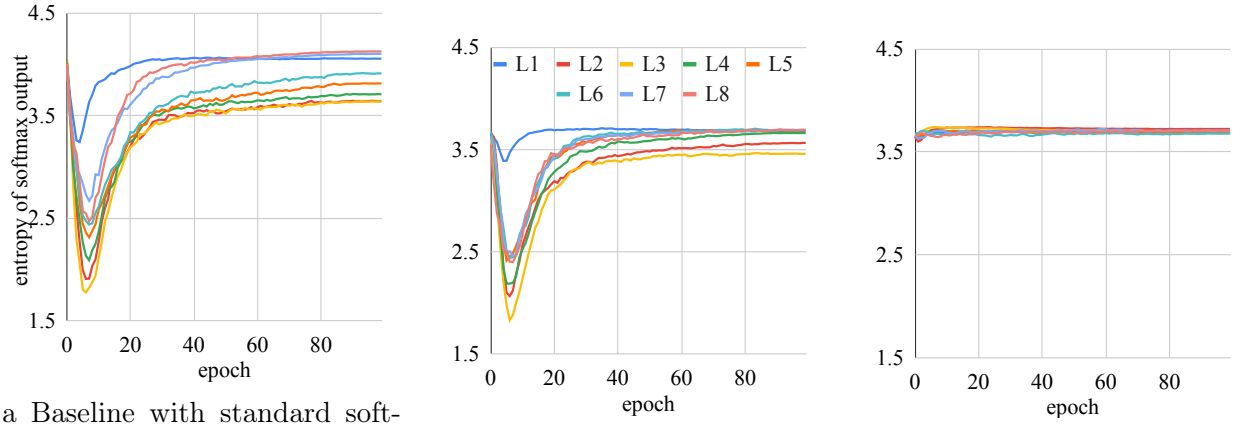
Acceleration. Figure 2.6a demonstrates the result with different training epochs.

When the number of epochs is small, NormSoftmax can achieve a better test accuracy than the standard softmax function. When we train with more iterations, NormSoftmax performs similarly to the baseline. The larger pre-defined threshold γ may translate into a higher acceleration, which pushes the model to escape the initial distraction stage more quickly. That is why `NSM-inf` performs better than `NSM-sqrtd` when the number of epochs is small. However, the normalization has a strict regularization effect on the softmax input, which impedes the training in the intermediate and final stages. Hence, the `NSM-inf` is exceeded by `NSM-sqrtd` when the number of training iterations is large. In short, NormSoftmax can accelerate the training process without sacrificing the representation ability, similar to curriculum learning [196].

Stabilization. We investigate the role of each component in the training recipe, with results listed in Figure 2.6b. We conduct ablation studies by removing (1) learning rate warmup, (2) weight decay, (3) label smoothing, and (4) strong data augmentation (random erasing, mixup, cutmix, and TrivialAugment) separately. We also replace the default AdamW optimizer with stochastic gradient descent with momentum (SGDM). The results indicate that the techniques above are critical to Transformer no matter what softmax function we use. However, Transformer with standard softmax and scaling factor $d^{-1/2}$ is more sensitive to the training techniques that are related to optimization. Without weight decay, the test accuracy of the baseline degrades from 87.96% to 84.15%, while the `NSM-inf` has a small drop from 88.01% to 87.46%. They share the same robustness against the training recipe for data augmentation.

We also alter the hyperparameters of the ViT and list the results in Figures 2.6c and 2.6d. The three methods share similar results when the depth or the hidden dimension is

small. However, large depth and hidden dimension impose a challenge for training. `NSM-inf` is more robust and provides much better results than the standard softmax.



a Baseline with standard softmax

b NormSoftmax with $\gamma = \sqrt{d}$

c NormSoftmax with $\gamma = +\infty$

Figure 2.7: NormSoftmax reduces the information entropy of softmax output in the 8-layer ViT.

We plot the information entropy of softmax output in this ViT in Figure 2.7. In baseline, the large standard deviation of softmax input induces small information entropy in the softmax output. The "distraction-concentration-distraction" effect is visualized in Figure 2.7a. The attended areas of Transformer undergo significant change during the training process. On the other extreme, `NSM-inf` generates the output, whose information entropy only slightly fluctuates since the input variance is always 1. The small change in the entropy of the output contributes to the acceleration and stability of the initial training process. The `NSM-sqrtd` is an interpolation between these two extremes.

Variants of NormSoftmax. We define several variants of NormSoftmax. (1) Adding learnable parameters for affine transformation defined by $f_1(\boldsymbol{x}) = \text{softmax}\left(w\boldsymbol{x}/\sigma(\boldsymbol{x})\right)$,

where $w \in \mathbb{R}$ is a learnable parameter; (2) Inverting the NormSoftmax defined by $f_2(\boldsymbol{x}, \gamma) = \text{softmax}\left(\frac{\boldsymbol{x}}{\max(\sigma(\boldsymbol{x}),\gamma)}\right)$; (3) NormSoftmax with different γ values; and (4) Logit Normalization.

For the first variant f_1, we find that the learnable parameter is not necessary. As indicated by [202], the learnable weight may induce the risk of overfitting. The second variant f_2 has an inverse clipping and is worse than our proposed NormSoftmax. It cannot accelerate the training since it encourages distraction in the Vision Transformer. In the third variant, the γ is a critical hyperparameter. $\gamma = 1, \sqrt{d}/2$ is too small, and the input vector is not effectively scaled. $\gamma = \sqrt{d}, 2\sqrt{d}$ obtains similar result. For Logit Normalization, we sweep the parameter of temperature τ, and find that $\tau = n^{-1/2}$ is almost the best one. With this temperature, the only difference between LogitNorm and NormSoftmax is whether the input is shifted to zero-mean, as shown in Equation equation 2.26. The accuracy with this temperature is 87.72%, which is even worse than the baseline, demonstrating the importance of shifting.

Scaling factors in attention The default scaling factor in attention is $d^{-1/2}$. We sweep the scaling factors for the standard softmax. We apply the corresponding γ in NormSoftmax.

scaling factor for sm	1	$2d^{-1/2}$	$d^{-1/2}$	$d^{-1/2}/2$	d^{-1}	
test accuracy for sm	80.98	86.38	87.96	87.97	86.98	
γ for nsm	1	$d^{1/2}/2$	$d^{1/2}$	$2d^{1/2}$	d	∞
test accuracy for nsm	81.04	87.16	88.13	88.21	87.1	88.01

Table 2.6: Results for (1) the standard softmax (sm) with different scaling factors and (2) NormSoftmax (nsm) with different γ

Table 2.6 shows that scaling factors of $d^{-1/2}/2$ and $d^{-1/2}$ achieve the best performance

for the standard softmax, similar to [107]. NormSoftmax shares the same trend with the standard softmax, achieving the best performance with γ of $2d^{1/2}$ and $d^{1/2}$. NormSoftmax achieves better results than the standard softmax with the corresponding pair of scaling factor and γ.

Temperature after normalization in NormSoftmax. We add a temperature parameter τ in NormSoftmax.

$$\text{NormSoftmax}(\boldsymbol{x}, \gamma, \tau) = \text{softmax}\left(\frac{\boldsymbol{x}}{\tau \min(\sigma(\boldsymbol{x}), \gamma)}\right) \tag{2.27}$$

If $\gamma = +\infty$, the input of softmax will always have a standard deviation of τ^{-1}. For a finite γ, we have $\sigma\left(\frac{\boldsymbol{x}}{\tau \min(\sigma(\boldsymbol{x}),\gamma)}\right) \geq \tau^{-1}$

Figure 2.8 illustrates the results with different temperature parameters. $\tau = 1$ is the best choice for both `NSM-inf` and `NSM-sqrtd`. When $\tau < 1$, `NSM-inf` is much more stable than `NSM-sqrtd` since the softmax input may have a high variance for `NSM-sqrtd`. On the contrary, when $\tau > 1$, `NSM-sqrtd` is better than `NSM-inf`. The softmax input of `NSM-inf` has a low standard deviation, restricting the performance.

Other benchmarks. We follow the reference implementation provided by PyTorch [142] to train different ViTs on ImageNet [43] from scratch. Strong data augmentations and many techniques are adopted. [8] Results are listed in Table 2.6. NormSoftmax can achieve better performance with a small number of epochs and similar performance with 300-epoch training.

We conduct experiments on machine translation with Transformers following the settings in [202]. The benchmarks are WMT English-German Translation (en-de), IWSLT

[8] The implementation is at this link.

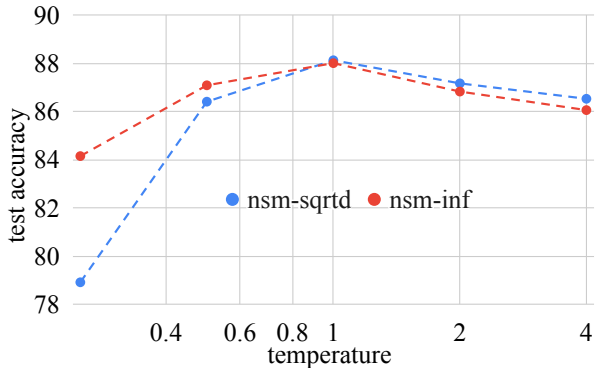


Figure 2.8: Test accuracy of a ViT with different temperatures after normalization.

	100 epochs			300 epochs		
	sm	nsm-sqrtd	nsm-inf	sm	nsm-sqrtd	nsm-inf
ViT-B-32	71.40	72.01	**72.64**	75.91	75.92	**75.95**
ViT-L-32	72.54	**73.49**	73.46	76.97	**77.01**	76.92
ViT-B-16	76.52	77.01	**77.10**	81.07	81.05	**81.09**
Swin-T	76.91	**77.50**	77.42	81.41	**81.52**	81.45

Table 2.7: Test accuracy of three ViT variants trained on ImageNet-1K from scratch.

2014 German-English Translation (de-en), and IWSLT 2015 English-Vietmanese Translation (en-vi) [27]. We replace the standard softmax function in both encoder and decoder with our proposed NormSoftmax. The evaluation metric is BLEU [141]. Similar to the results in computer vision, NormSoftmax can also accelerate the learning process in natural language processing.

	45k steps			90k steps		
	sm	nsm-sqrtd	nsm-inf	sm	nsm-sqrtd	nsm-inf
en-de	24.2	25.6	25.5	28.3	28.3	28.2
de-en	30.1	30.9	31.5	35.4	35.5	35.4
en-vi	26.7	27.1	27.2	31.2	31.3	31.4

Table 2.8: The BLEU on three machine translation benchmarks with Transformers.

2.2.4.2 Cross Entropy Loss of the Classification Problem

We follow the example in the JAX framework [19] to train ResNets [63] on ImageNet, which has 1,000 classes and about 1.3 million training images. [9] We use SGDM with linear warmup and cosine annealing learning rate scheduler, accompanied by a large mini-batch size of 8,192 and a large learning rate of 3.2. We only enable the horizontal flip and input normalization as data augmentation techniques. We set γ as 1 and $+\infty$ in NormSoftmax for the cross entropy loss since 1 is the temperature in the baseline. We also apply Logit Normalization and compare it with our proposed method. For Logit Normalization, we conduct a grid search to find the optimal hyperparameter of temperature.

Figure 2.9 and Table 2.9 show that the NormSoftmax can boost the initial training. With 20 epochs, `NSM-inf` can achieve the test accuracy of 71.42% while the baseline with standard softmax obtains 69.18%. With sufficiently long epochs, `SM` and `NSM` achieve similar test accuracy, implying that `NSM` does not impede the representation learning ability of the models. With larger pre-defined threshold γ, `NSM-inf` is faster than `NSM-1`. NormSoftmax achieves better results than Logit Normalization, demonstrating that it is necessary to shift

[9] The implementation is available at this link.

the input vector before applying Logit Normalization.

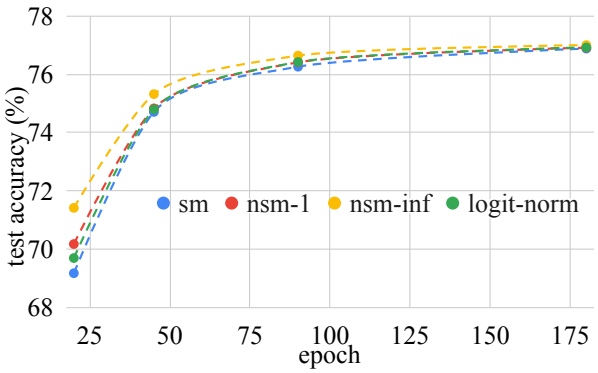


Figure 2.9: Test accuracy of ResNet50 ImageNet-1K with different training epochs

Training performance of a deep ResNet.

We train a ResNet1202, which is investigated in the original ResNet paper [63]. Figure 2.10 compares the test accuracy during the training process. NormSoftmax can significantly accelerate the training process and achieve better test accuracy than the standard softmax. With standard softmax, the model is stuck in the low confidence zone in the initial stage.

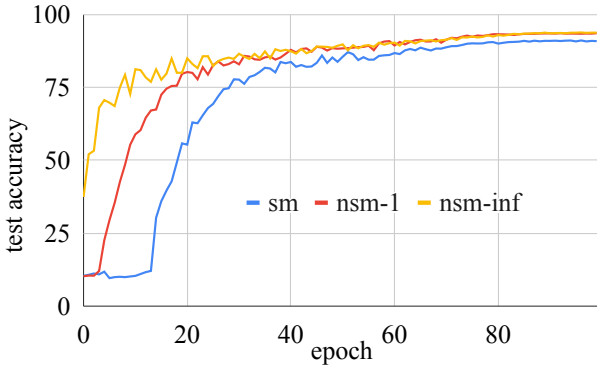


Figure 2.10: Test accuracy of a ResNet1202 on CIFAR-10

# epochs		20	45	90	180
R50	sm	69.18	74.71	76.26	76.88
	nsm-1	70.18	74.83	76.41	76.91
	nsm-inf	71.42	75.32	76.64	77.17
	lognorm	69.70	74.79	76.42	76.92
R101	sm	72.44	76.21	77.79	77.99
	nsm-1	72.48	76.18	77.80	78.08
	nsm-inf	72.69	76.56	78.00	78.13
	lognorm	72.50	76.19	77.82	78.01
R152	sm	72.79	76.78	78.18	78.51
	nsm-1	72.80	76.85	78.32	78.50
	nsm-inf	72.60	76.73	78.17	78.44
	lognorm	72.75	76.81	78.15	78.40

Table 2.9: Test accuracy on ImageNet with ResNets. *lognorm* is short for Logit Normalization.

2.2.5 Summary

We investigate the behavior of softmax in neural network training and discuss its impacts on training stability and convergence. We find that one of the reasons for the optimization difficulty is the significant change in the variance of softmax input during the early training process. To remedy the optimization difficulty of softmax, we propose a simple yet effective substitution, named NormSoftmax, where the input vectors are first re-scaled by dynamically calculated vector-specific factors and then fed to the standard softmax function. Similar to other existing normalization layers in machine learning models, NormSoftmax can stabilize and accelerate the training process and also increase the robustness of the training procedure to hyperparameters. Experiments on Transformers in computer vision and natural language processing benchmarks validate that our proposed NormSoftmax is an effective

plug-and-play module to stabilize and speed up the optimization of neural networks with cross-entropy loss or dot-product attention operations.

Limitations. Transformer-based models require large amounts of data, and their performance is constrained when trained on small or intermediate datasets. The full representation potential of these models can be realized with larger datasets. In this study, we only present results obtained from small or medium-sized datasets. The training behavior on larger datasets may differ from our observations on smaller ones. Additionally, we find that NormSoftmax may have a minor impact on the final performance, primarily aiding in the acceleration and stabilization of the training process.

2.3 Mixed Precision Neural Architecture Search for Energy Efficient Deep Learning

While most of the inferences currently reside in the cloud, it is increasingly desirable to deploy the trained DNNs to edge devices, such as mobile phones and wearable devices, due to privacy, security, and latency concerns or limitations in communication bandwidth. The increasing gaps between complicated DNNs and hardware implementations deteriorate the edge inference [203]. Hence, the growing demand for small-size and energy-efficient DNNs is motivated to meet the limited area and energy budget of edge applications.

Model quantization. Research has shown that there are redundancies in both the number of weights and the number of bit representations (or precisions) of weights and Arithmetic in DNNs [155]. Neural networks can be compressed using weight clustering and quantization to reduce the computation complexity with negligible loss of accuracy. In the quantization approach, full precision floating point representation is replaced by lower fixed

point precision or even binary representation to achieve a significant reduction in computation as well as energy consumption. Learning layer-wise bitwidths has been proposed recently [185, 194] to explore the search space of layer-wise quantization policies.

Neural Architecture Search (NAS). Another major approach to learning energy efficient deep networks is by designing efficiency-friendly neural architectures [24,28,200,221]. Recently, researches show that automatically designing neural architectures by efficiency-aware NAS [24, 61, 193] can bring more benefits than hand-crafted design. Following NAS framework, compact and efficient architectures are found by either searching from scratch [24, 193], or pruning or distilling from well trained large neural networks [46, 155, 189].

However, the existing works focus on either of the two methods aforementioned while neglecting the potential of their interaction and joint optimization for further improvement. Intuitively, the optimal choices of bit-widths and architectures are correlated. For instance, in MobileNets [161], one should retain more bits for the bottleneck layers (the expansion and projection layer) which encode the model's intermediate inputs and outputs using small 1×1 convolutional filters; on the other hand, one may consider using fewer bits in the depthwise convolution layers because empirically, they are often over-parameterized, memory bounded and are less sensitive than bottleneck layers. The combination of NAS and quantization contributes to the final accuracy and energy efficiency of neural architectures in a complex and interleaved manner. Therefore, a synergistic co-optimization of NAS and quantization is likely to allow us to achieve more hardware efficient solutions.

2.3.1 Overview

We propose to learn more hardware-efficient deep networks by co-optimizing both precision and NAS. Compared with the hand-crafted heuristic design which often falls in sub-optimal results, our method can achieve better solutions. Our algorithm leverages a differentiable neural architecture search method with Gumbel-Softmax to directly search the optimal combinations of precision and architectures, to minimize both the accuracy and the energy cost, estimated from a physical simulator. We conduct extensive studies on two widely-adopted image classification benchmarks, CIFAR-100 [98] and ImageNet2012 [158], on which significant improvements are obtained. The contributions of this work are highlighted as below.

- Our approach yields more energy efficient deep learning by co-optimizing the neural network architectures and quantization policies that assign different precision to different blocks of the network. To our best knowledge, this work is the first one to explore the end-to-end co-optimization of NAS and mixed precision quantization.

- We develop a framework to effectively search the new solution space. We train a quantized DNN during the search, and adopt the REINFORCE algorithm for the non-differentiable energy oracle from hardware simulators.

- Our experimental results show that the co-optimized architectures and the bitwidths settings can achieve lower error rate and less energy consumption on CIFAR-100 and ImageNet2012 tasks than strong baseline approaches. Specifically, in ImageNet2012, we can reduce 63% energy with almost no loss of top-1 accuracy, compared with 8-bit MobileNetV2.

2.3.2 Related Work

In this section, we first introduce a general topic, hardware-aware machine learning. Then we discuss two related topics: searching hardware-friendly neural network architectures and quantization.

2.3.2.1 Hardware-Aware Machine Learning

With the huge success of deep learning, there is an increasing demand for pushing it to the edge. However, DNNs are quickly evolving towards deeper and more complicated architectures for higher accuracy [203]. Hence, it is energy hungry and less run-time efficient on the edge inference, raising the necessity of hardware-aware machine learning design.

Besides the hardware-aware compression and quantization, researchers also pay attention to the design of efficient neural architectures for hardware-aware machine learning [128].

Some efficient neural architectures have been designed hand-crafted, e.g. *MobileNet* [161], *ShuffleNet* [213]. Others are developed automatically by neural architecture search methods [24, 193].

2.3.2.2 Neural Architecture Search (NAS)

NAS [49] has demonstrated superior performance on many challenging applications, such as image classification [221], object detection [193], natural language processing [28], to name a few. Most pioneer works in NAS [28, 122, 171, 200, 221] focused on searching novel architectures such that the task-oriented performance (usually accuracy) is optimized without taking hardware performance into consideration. Some of the previous works [61, 171, 184] try to find efficient network architectures, but mainly focus on the latency on GPU

and CPU.

Some NAS related works have also been done on searching efficient neural networks for edge devices. *FBNet* [193] focuses on NAS with less latency on mobile devices, which relaxes the non-differentiable discrete space to differentiable continuous space using Gumbel-Softmax [80]. *ProxylessNAS* [24] also focuses on NAS for given mobile devices, where the binarized parameters are trained based on *BinaryConnect* [37]. Both works use MobileNetV2 block [161] as the base search unit, which is proved energy efficient and low-latency on edge devices.

Weight pruning can be viewed as a type of NAS, which discovers a small neural network from an over-parameterized neural network [119]. Weight pruning leverages the inherent redundancy in the number of weights, thereby achieving effective model compression with negligible accuracy loss. However, its irregular sparsity and complicated indexing scheme may induce overhead in hardware implementation and require careful optimization [46,155,189]. In this work, we focus on other schemes that are more friendly to hardware implementation, but our methodology can be extended to include pruning as well.

2.3.2.3 Quantization

Model quantization methods remove the redundancies in the neural networks and bit representations, and they are able to reduce the computational complexity significantly. The quantized models offer the potential of remarkable memory and computation efficiency, while achieving the accuracy of their full-precision counterparts [62, 97]. Moreover, weight quantization, especially equal-distance quantization, is more hardware friendly than weight pruning methods.

The critical point for quantization is to keep the balance of task-specific performance and hardware-related metrics.

Some work tries to train a binary neural network, which can extremely save energy, latency, and other hardware-related metrics at the cost of severe degradation on accuracy. *BinaryConnect* and *XNOR Net* binarize all the weights and activations [37, 152]. However, it causes an extreme loss on accuracy. For instance, applying *XNOR Net* on AlexNet [99] achieves 44.27% top-1 accuracy on ImageNet2012, which is far worse than the full-precision accuracy 56.6%. Other works focus on training low-precision neural networks, which can keep the balance between efficiency and accuracy better than binary neural networks. *Deep Compression* quantizes the network weights to reduce the model size by rule-based strategies [62]. Jacob et al. and Banner et al. train neural networks with 8-bit precision [7, 77], with straight through estimator and range batch normalization.

However, once the neural network becomes deeper, the search space of layer-wise bitwidth increases exponentially, which makes it infeasible to rely on hand-crafted strategies. The heuristic layer-wise bitwidths are believed to be sub-optimal [185], and cannot keep the balance between accuracy and efficiency. Recent works try to search layer-wise bitwidths for a pre-trained model with a particular architecture. *HAQ* [185] searches layer-wise bitwidths for MobileNet using DDPG [114], and adds a fine-tune process with few iterations after quantization. Guo et al. develops a evolutionary algorithm to search layer-wise bitwidths [61]. *Mixed Precision Quantization* [194] and *Stochastic Layer-Wise Precision* [102] search with Gumbel-Softmax [80] are not tested on some efficient neural architectures, e.g. MobileNet [161], ShuffleNet [213]. Applied on a pre-trained model, all these works can achieve better accuracy than one single bit-width and heuristic layer-wise bitwidths. These

empirical results show that automatically searching the layer-wise bitwidth is helpful for deep neural network quantization.

2.3.3 Algorithm

In this section, we introduce the general form of our algorithm and present our joint NAS and mixed precision quantization framework.

2.3.3.1 Energy Constrained NAS

Figure 2.11 shows the overall flow of our model. We aim at searching novel energy efficient neural architectures. Our method can be viewed as a *controller* that interacts with a *task environment* (e.g. image classification task) and a *hardware environment* (e.g. physical energy simulator). The goal of the controller is to discover novel neural architectures that minimize the task-related loss while satisfying some energy constraints. To be concrete, our training objective can be written as follows,

$$\begin{aligned} &\min_{\theta} \mathbb{E}_{\alpha\sim\pi_\theta}\Big[\mathcal{L}(f_{w^*}(\alpha);\ \mathcal{D}^{val})\Big], \\ \text{s.t.}\ \ &w^* = \arg\min_{w} \mathcal{L}(f_w(\alpha);\ \mathcal{D}^{trn}, \alpha), \\ &\mathbb{E}_{\alpha\sim\pi_\theta} J(\alpha) < c. \end{aligned} \tag{2.28}$$

The NAS controller π_θ generates *samples* $\{\alpha\}$ that are different configurations of neural networks. Given a network configuration α, $f_w(\alpha)$ defines a deep neural network model associated with model weights $w(\alpha)$. In the case of image classification, network f_w could be the AlexNet [99] that maps an image to a probability distribution over predicted output classes. $\mathcal{L}(\cdot)$ stands for the task-dependent loss (e.g., cross entropy for image classification)

and $J(\alpha)$ measures the energy cost of the network initialized by a configuration α. The controller is trained to find the best internal parameters θ so that it will generate architectures that achieve the minimum expected task-specific loss $\mathcal{L}(f_{w^*}(\alpha); \mathcal{D}^{val})$ evaluated on a validation set $\mathcal{D}^{val}$, where the network weights $w^*(\alpha)$ are obtained by minimizing the loss function on a training set $\mathcal{D}^{trn}$. Meanwhile, the expected energy consumption is constrained to be smaller than a threshold c to promote energy efficient architecture search.

We first relax the energy constrained objective equation 2.28 as follows,

$$\min_{\theta} \mathbb{E}_{\alpha\sim\pi_\theta}\Big[\mathcal{L}(f_{w^*}(\alpha);\ \mathcal{D}^{val})\Big] + \lambda\Big[\mathbb{E}_{\alpha\sim\pi_\theta}[J(\alpha)] - c\Big]_+ \tag{2.29}$$

$$\text{s.t.}\quad w^* = \arg\min_{w} \mathcal{L}(f_w(\alpha);\ \mathcal{D}^{trn}, \alpha), \tag{2.30}$$

with $[x]_+ = \max(x, 0)$ and λ a hyper-parameter, which only penalizes the objective function when the energy constraint is violated.

However, it is still infeasible to solve the optimization problem equation 2.29 correctly. The computational difficulties arise in two-fold. First, the optimal model weights $w^*(\alpha)$ depend on model configuration α, an inner loop minimization is required whenever the control parameter θ is changed. Second, in general, energy-based objectives $J(\alpha)$ are non-differentiable, which prohibits the use of efficient back-propagation.

To solve the aforementioned challenges, motivated by DARTS [116], we approximate $w^*(\alpha)$ with one step gradient descent, specifically,

$$w'(\alpha) = w(\alpha) - \epsilon\nabla_w \mathcal{L}(f_w(\alpha); \mathcal{D}^{trn}, \alpha), \tag{2.31}$$

with ϵ as the step size. Without considering the energy related terms, our objective function

is reduced to

$$\min_{\theta} \mathbb{E}_{\alpha \sim \pi_\theta} \Big[\mathcal{L}(f_{w'}(\alpha);\ \mathcal{D}^{val}) \Big].$$

We then apply gradient descent for optimization, such that the model parameters θ are updated as follows:

$$\theta \leftarrow \theta - \beta \nabla_\theta \mathbb{E}_{\alpha \sim \pi_\theta} \Big[\mathcal{L}(f_{w'}(\alpha);\ \mathcal{D}^{val}) \Big] \tag{2.32}$$

where β is the step size. In practice, the expected gradient is approximated using Monte Carlo samples $\{\alpha_i\}_{i=1}^m$ drawn from π_θ, combining with Equation equation 2.31, we can approximate Equation equation 2.32 as follows,

$$\theta \leftarrow \theta - \beta \nabla_\theta \Big[\frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(f_{w - \epsilon \nabla_w \mathcal{L}(f_w(\alpha_i); D^{trn})}; \mathcal{D}^{val}) \Big]. \tag{2.33}$$

To deal with the non-differentiable energy measures, we use the REINFORCE algorithm [192]. The expected gradient can be computed as follows

$$\begin{aligned} \nabla_\theta \mathbb{E}_{\alpha \sim \pi_\theta} J(\alpha) &= \nabla_\theta \int J(\alpha) \pi_\theta(\alpha) d\alpha \\ &= \int J(\alpha) \nabla_\theta \pi_\theta(\alpha) d\alpha = \int J(\alpha) \pi_\theta(\alpha) \nabla_\theta \log \pi_\theta d\alpha \\ &= \mathbb{E}_{\alpha \sim \pi_\theta} \Big[J(\alpha) \nabla_\theta \log \pi_\theta \Big] \approx \frac{1}{k} \sum_{i=1}^{k} J(\alpha_i) \nabla_\theta \log \pi_\theta(\alpha_i). \end{aligned} \tag{2.34}$$

The full algorithm is outlined in Algorithm 2.

2.3.3.2 Mixed Precision Architecture Search Space

In order to search an energy efficient neural architecture and have a fair comparison with previous NAS algorithms [61, 193], we adopt the commonly used MobileNetV2 (MB)

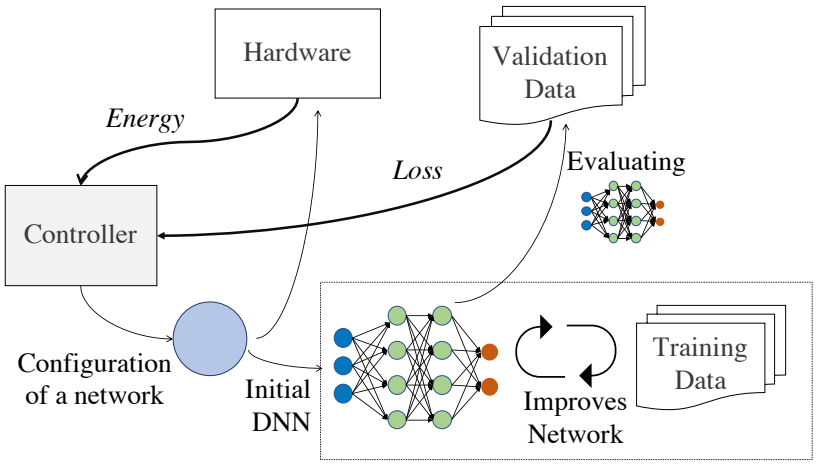


Figure 2.11: Illustration of our energy aware neural architecture search framework.

Algorithm 2 Energy aware neural architecture search

1: **repeat**
2: Sample minibatch of network configurations $\{\alpha_i\}_{i=1}^{k}$ from the controller π_θ
3: Update network models $\{w(\alpha_i)\}$ by minimizing the training loss following Eqn. 2.31.
4: Update the controller parameters θ following Eqn. equation 2.33 and equation 2.34.
5: **until** Converge

block [161] as the base search unit, which is demonstrated in Figure 2.12. Each MB block consists of two bottleneck layers (expansion and projection layer) and one depthwise convolution layer. First, the 1×1 expansion convolution layer expands the number of channels by a factor of m before the data goes into the depthwise convolution. Second, the depthwise convolution applies $k \times k$ filters to its input while keeping the number of output channels the same. Finally, a 1×1 projection convolution layer squeezes the network in order to match the initial number of channels. We allow each MB block with a filter size $k \in \{3, 5, 7\}$ and an expansion ratio $m \in \{1, 3, 6\}$.

We assume the final neural network architecture is hierarchical that stacks of a certain

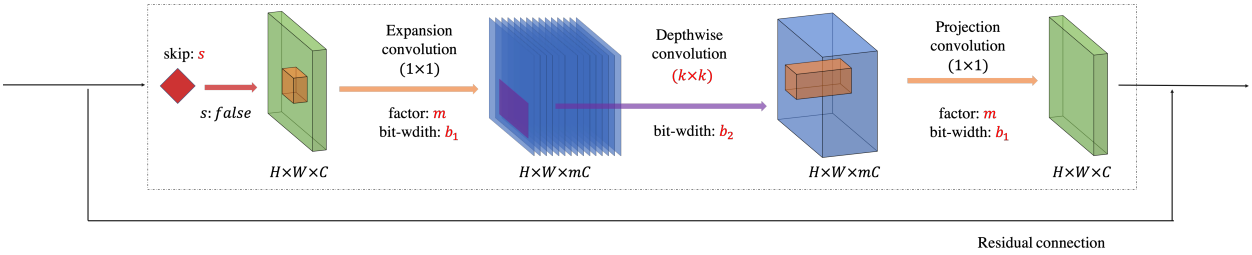


Figure 2.12: An illustration of the structure of a MB search unit. m denotes the expand ratio, k denotes the kernel size, s denotes whether skipping this block, and b_1, b_2 denotes the layer-wise bitwidths.

number of MB blocks. In order to search the number of MB blocks, we add a zero operation (skip connection, marked as s in Figure 2.12.) into the search space. The entire MB block will be skipped if the skip operation is selected (s: true).

In addition, we augment our NAS search space to allow each MB block to choose the quantization precision adaptively. Note that in a MB block the depthwise convolution layer normally contains more parameters than the bottleneck layers, and thus more energy hungry. We propose to assign two different bitwidths for each MB block, and search the optimal precision choices for the bottleneck layers (b_1) and the depthwise layer (b_2), respectively. And $b_1, b_2 \in \{2, 4, 6, 8\}$ bits. Table 2.10 shows the macro-architecture of our search space.

2.3.3.3 Search Algorithm

We are now ready to show the representation of the network configuration α. Let $\eta_\ell = [s^\ell, m^\ell, k^\ell, b_1^\ell, b_2^\ell]$ be the discrete variables that associated with the MB block in the ℓ-th layer. A network configuration α is a stack of N MB blocks as a vector of $[\eta_1, ..., \eta_N] \in \mathbb{R}^{5N}$, where N denotes the number of MB blocks. Each element in α can be considered as a specific operation. Without loss of generality, denote $o(x)$ as one of the operator defined in α, which

Input Shape	Block and Bit-width	#channels	stride	n
$224 \times 224 \times 3$	Conv3 $\times$ 3, 8bit	32	2	1
$112 \times 112 \times 32$		16	2	1
$56 \times 56 \times 16$		24	1	2
$56 \times 56 \times 24$		32	2	4
$28 \times 28 \times 32$	MB(s, m, k, b_1, b_2)	64	2	4
$14 \times 14 \times 64$		128	1	4
$14 \times 14 \times 128$		160	2	5
$7 \times 7 \times 160$		256	1	2
$7 \times 7 \times 256$	Conv1 $\times$ 1, 8bit	1280	1	1
$7 \times 7 \times 1280$	Pooling & FC, 8bit	-	1	1

Table 2.10: This table shows the macro-architecture of the search space. and n denotes the number of repeated MB layers with the same number of channels. (s, m, k, b_1, b_2) stands for MB block configurations.

takes K possible values. E.g., $o(x)$ could be a depthwise convolution layer, in this case, all possible filter size choices are $\{3 \times 3, 5 \times 5, 7 \times 7\}$ and $K = 3$. Given the input x, $o(x)$ maps x to its corresponding output.

To learn the controller, it remains to show the representation of π_θ and the gradient of the validation loss $\nabla_\theta \mathbb{E}_{\alpha \sim \pi_\theta}[\mathcal{L}(f_{w'}(\alpha), \mathcal{D}^{val})]$ w.r.t. θ defined in Eqn. 2.33. For each operator $o(x)$, we represent discrete valued architecture choices as a one-hot vector $d \in \mathbb{R}^K$, such that $d_j \in \{0, 1\}$ and $\sum_{j=1}^{K} d_j = 1$. In this way, the corresponding output of applying operator o with representation d is $\sum_{j=1}^{K} d_j o_j(x)$, where $o_j(x)$ denotes the forward operation with j-th candidate in the search space (e.g., o_1 could be 3×3 depthwise convolution). We parametrize $p(d_j = 1) = \exp(\phi_j)/\sum_{i=1}^{K} \exp(\phi_i)$, where $\{\phi_j\}$ are trainable parameters that belongs to the controller parameters θ. It is problematic to calculate the gradients directly due to the inability to back-propagate through categorical samples. We thus replace

the non-differentiable network configuration samples with continuous differentiable samples from a Gumbel-Softmax distribution [80]. That is, we replace one-hot representation d as a continuous vector such that $d_j \geq 0, \sum_{j=1}^{K} d_j = 1$, which can be trained using standard back-propagation. For the rest, we follow the settings for Gumbel-softmax sampling used in FBNet [193].

Training with Quantized Weights. In our approach, only the quantized values of the weights and activations are used in all forward operations. In order to learn quantized weights, we follow the linear quantization schemes suggested by [155, 185]. Specifically, for a layer with $n \times d$ dimensional input $\boldsymbol{x} = (x_1, ..., x_n), x_i \in \mathbb{R}^d$, we quantize each x_i linearly into b bits:

$$\text{Quantize}(x_i, b) = \text{round}(\text{clamp}(x_i, c)/s) \times s,$$

where clamp truncates all values into the range of $[-c, c]$, with $c = \max\{|x_i|\}$. And the scaling factor s is defined as $s = c/(2^{b-1} - 1)$.

2.3.4 Experimental Results

We search architectures and mixed quantization precision for each layer on a proxy task, tiny ImageNet [10]. Next, we train the discovered architectures on two image classification benchmarks (CIFAR-100 and ImageNet) from scratch. We show that our method achieves competitive (or better) accuracy compared with state-of-the-art deep neural architectures and simultaneously yields significant smaller model size and lower energy consumption.

Low precision neural network baselines are quantized and then finetuned using the

[10] https://tiny-imagenet.herokuapp.com/

standard compression methods proposed in [62] unless otherwise specified.

2.3.4.1 NAS

Energy Modeling. We use the simulator for Bit Fusion [11] [164] to obtain the hardware-related performance metrics. Bit Fusion employs a 2D systolic array of bit-level processing elements that dynamically fuse to match the bitwidth of every single layer. For a fair comparison, we exactly follow the setting described in HAQ [185] for energy estimation. All the energy consumption discussed below are for the inference with a batch of 16 images.

Settings. The tiny ImageNet dataset consists of 200 classes. Each class contains 500 training images, 50 validation images, and 50 test images, respectively. For preprocessing, we resize the image to the size of 224×224. We use a batch size of 64 and adopt label smoothing for training. Our algorithm is trained for a fixed 60 epochs during the architecture search process. We run our search program on one NVIDIA Tesla P100 GPU, which takes five days.

Results. Note that we jointly optimize the task-oriented loss and energy constrains, the trade-off between these two terms are controlled by a hyperparameter λ (defined in Equation 2.29). Large λ values lead to smaller energy efficient models while less accurate on the prediction task; vice versa. We evaluate two different settings $\lambda = 0.1, 0.01$, which result in one smaller and one bigger neural architecture (see Figure 2.13), respectively. We can see from Figure 2.13 that depthwise convolution layers (more parameters) are allocated with fewer bitwidths than bottleneck convolution layers (fewer parameters), as also observed in HAQ [185]. In addition, we notice that bottleneck layers with small expand ratios (fewer

[11] https://github.com/hsharma35/bitfusion

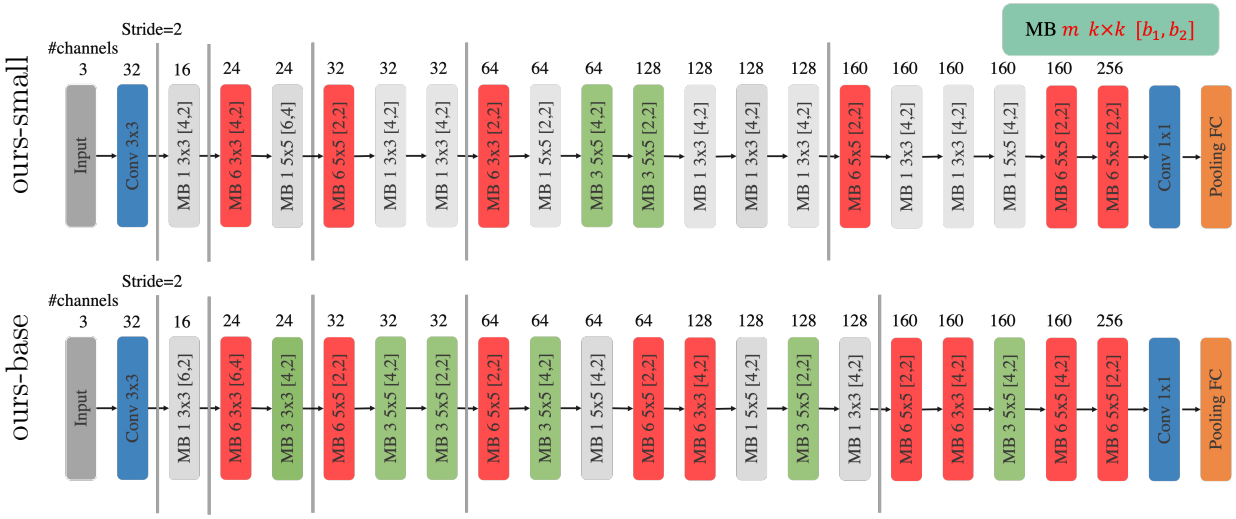


Figure 2.13: Two energy efficient deep neural architectures found by our method. "MB m $k \times k$ $[b_1, b_2]$ " represents a mobile inverted bottleneck convolution layer, for which m stands for the expansion ratio, $k \times k$ is the filter size for the the depthwise convolution layer, b_1 and b_2 represents the bit-width precision for the bottleneck layers (expansion and projection convolution layers) and the depthwise layer, respectively.

parameters) tend to have larger bitwidths to preserve good accuracy.

2.3.4.2 Experiments on ImageNet

Dataset. The ILSVRC 2012 classification dataset [43] consists of 1.2 million training images and 50,000 validation images, with 1,000 classes. We resize the image size to 224×224, and adopt the standard data augment scheme (mirroring and shifting) for training images.

Settings. For ImageNet, we set the batch size to be 64, and fix the number of the filters in the first convolution layer to be 32. We use stochastic gradient descent with an initial learning rate 10^{-4} and apply cosine learning rate annealing scheduling [64]. We also use label smoothing [175] ($\alpha = 0.1$), mixup [211] ($\alpha = 0.2$) for data augmentation, and clip the gradient to the range of $[-5, 5]$. We replace the vanilla batch normalization layer with 8-bit range normalization following [7], and train our discovered neural architectures for 120 epochs on the training set.

Model	Precision	Top-1 Error (%)	Top-5 Error (%)	Model Size (Bytes)	Energy (mJ)	Latency (ms)
VGG-16 [167]	8-bit	29.10	7.40	138.00M	753.11	838.03
ResNet-50 [63]	8-bit	24.70	5.30	25.50M	557.46	591.17
MobileNetV2 [185]	8-bit	28.19	9.75	3.40M	29.01	73.85
FBNet-B [193]	8-bit	26.84	8.97	4.50M	34.65	83.91
FBNet-B [193]	3-bit	36.29	15.38	1.68M	13.47	27.93
HAQ-small [185]	*mixed*	33.01	12.67	1.70M	12.85	32.10
ours-small	*mixed*	**31.62**	**11.56**	**1.44M**	**8.91**	**21.19**
HAQ-base [185]	*mixed*	29.10	10.09	2.12M	16.31	40.21
Ours-base	*mixed*	**28.23**	**9.94**	**2.06M**	**10.85**	**24.71**

Table 2.11: Results on the ImageNet2012 dataset.

Results. The performance of our models (denoted as *ours-small* and *ours-base*) is reported in Table 2.11 along with other state-of-the-art approaches. We report the top-1 and top-5 classification error on the validation set. In addition to the energy consumption, we also

show the latency and model size off all evaluated approaches. As shown in Table 2.11, our discovered models can extremely reduce the energy while achieving on par (better) accuracy compared to strong baseline approaches.

In the first block, we can see that classic DNNs are most energy hungry. For example, VGG-16 [167] costs 753.11mJ energy along with 838.03ms latency, ResNet-50 [63] costs 557.46mJ energy along with 591.17ms latency. The energy consumption of these two models is about 50× higher than our models. To have a fair comparison with HAQ [185] with similar level of energy consumption, we compare with two variants of HAQ, *HAQ-small* and *HAQ-base*: *HAQ-small* denotes the energy-conserving setting with more aggressive quantization strategies; *HAQ-base* denotes the setting favors better accuracy and thus larger models. Compared to *HAQ-small*, *ours-small* reduces the top-1 error rate from 33.01% to 31.62% and reduces energy by 3.94mJ. *Ours-base* also achieves lower top-1 error rate (30.60% → 28.23%) and leads to about 3× lower energy consumption (30.60mJ→ 10.85mJ) than *HAQ-Base*.

Compared to 8-bit MobileNetV2, *ours-base* achieves 35% reduction on model size and 63% reduction on energy, respectively. Compared with 3-bit FBNet-B [193], which has similar energy cost to *ours-base*, we can improve the top-1 error rate from 36.29% to 28.23%. Besides, our discovered models can be fitted into on-chip SRAM cache (5pJ per access under 45nm CMOS technology) rather than off-chip DRAM memory (640pJ per access under 45nm CMOS technology). Shown in Table 2.11, the model size of *ours-small* is 1.44MB, which could be accommodated into on-chip SRAM cache.

2.3.4.3 Experiments on CIFAR-100

Dataset. CIFAR-100 [12] consists of images drawn from 100 classes. The training and test sets contain 50,000 and 10,000 images respectively. We adopt a standard data argumentation scheme (mirroring and shifting) that is widely used for this dataset.

Settings. We set the batch size to be 128, set the learning rate to be 0.1, remove label smoothing and remove the first two down-sample convolutions in the architecture. We follow the other settings in the ImageNet task.

Model	Precision	Model Size (Bytes)	Error (%)	Energy (mJ)	Latency (ms)
DenseNet-BC-190 + Mixup [211]	8-bit	26.0M	17.02	125.74	247.21
ENAS + Cutout [145]	8-bit	4.6M	16.58	25.93	49.78
NAO + Cutout [122]	8-bit	10.8M	**15.87**	37.20	75.92
MobileNetV2	8-bit	2.5M	21.85	5.09	13.13
FBNet-B [193]	8-bit	2.8M	21.36	8.27	17.48
HAQ-small [185]	*mixed*	0.6M	22.93	1.17	2.75
ours-small	*mixed*	0.6M	**22.16**	**1.00**	**2.53**
HAQ-base [185]	*mixed*	0.8M	21.89	1.53	3.31
ours-base	*mixed*	0.8M	**21.27**	**1.21**	**2.98**

Table 2.12: Results on the CIFAR-100 dataset.

Results. The main results on CIFAR-100 are shown in table 2.12, we can see that pioneer state-of-the-art architectures, e.g. 8-bit DenseNet [71], lead to significant energy and memory cost, which is 125.74mJ and 26.0MB respectively. Recent NAS frameworks only focus on accuracy optimization that also results in architectures with notable energy consumption. For instance, 8-bit NAO [122] uses 37.20mJ; 8-bit ENAS [145] costs 25.93mJ. On the other hand, *ours-small* achieves $\times 5$ reduction in model size and more than $\times 5$ reduction

[12] https://www.cs.toronto.edu/~kriz/cifar.html

in energy consumption while achieving better accuracy compared to the 8-bit MobileNetV2. Compared to HAQ, *ours-small* costs less energy than HAQ-small, meanwhile, improves the error rate from 22.93% to 22.16%. *Ours-base* can achieve 0.32mJ energy reduction than HAQ-base, and improve the error rate from 21.89% to 21.27%.

Optimization	Model Size (Bytes)	Error(%)	Energy (mJ)
NAS+Quantization (Small)	0.5M	22.34	1.04
ours-small	0.5M	**22.16**	**1.00**
NAS+Quantization (Base)	0.9M	21.58	1.28
Ours-base	0.8M	**21.27**	**1.21**

Table 2.13: Comparison of stepwise NAS and model quantization v.s. our joint-optimizing framework. The models are evaluated on the CIFAR-100 dataset.

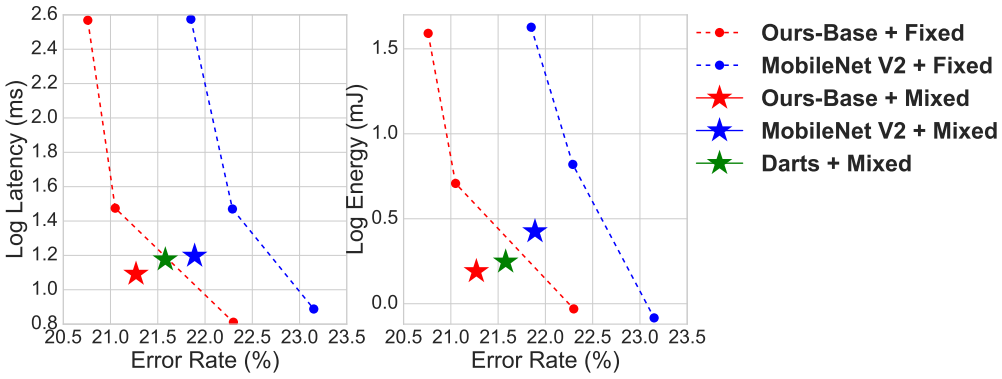


Figure 2.14: Comparison of mixed precision quantization (denoted as 'mixed') v.s. constant precision quantization (denoted as 'fixed'). The dashed line shows the results for $\{8, 4, 2\}$ bitwidths (left to right), respectively.

2.3.4.4 Joint NAS and Adaptive Mixed Precision Quantization

We further study the importance of joint NAS and mixed precision quantization. On the CIFAR-100 dataset, we study a baseline approach by performing NAS and model

quantization stepwisely. Specifically, we use DARTS [116] [13] for architecture search with the same search space defined in Table 2.10 while using 8-bits precision for all layers. Next, we perform layer-wise model quantization following HAQ. In a way similar to our approach, we set $\lambda = 0.1, 0.01$, respectively, and discover two neural models (NAS+Quantization (small) and NAS+Quantization (base)) with similar model size compared to *ours-small* and *ours-base*.

As we can see from Table 2.13, compared to *NAS+Quantization (small)*, *ours-small* reduces the error rate from 22.34% to 22.16% and also reduce the energy cost by around 0.05mJ. Compared to *NAS+Qunatization (base)*, *Ours-base* reduces the error rate from 21.58% to 21.27% and achieves a 5% reduction on energy consumption. These results highlights the benefits of joint NAS and mixed precision quantization as suggested by our approach.

In Figure 2.14, we show the advantage of adaptive mixed precision quantization versus constant precision quantization. In Figure 2.14, the dashed line shows the performance (latency v.s. classification error rate) of different baseline architectures with a constant 8-bit, 4-bit, and 2-bit precision, respectively. The star points show the results of HAQ and our method, that both allow flexible layer-wise bitwidths. With mixed layer-wise precision searching, both HAQ and our method discovers neural architectures yields good trade-off between accuracy and latency, and our method outperforms HAQ.

[13] https://github.com/quark0/darts

2.3.5 Summary

We propose a new methodology to perform NAS and mixed precision quantization in the extended search space with hardware performance involved in the objective function. Experimental results demonstrate that our proposed approach achieves better energy efficiency than advanced quantization approaches on CIFAR-100 and ImageNet. Our methodology facilitates the end-to-end design automation flow of neural network design and deployment, especially for edge inference. As future directions, we may extend this work to other neural architecture search methods, quantization techniques, and backbone machine learning models.

2.4 Summary of the Chapter

We present three methods to improve the efficiency of machine learning algorithms, including (1) effective gradient matching for dataset condensation, which enhances data efficiency; (2) NormSoftmax, improving training efficiency for softmax-based models; and (3) mixed precision NAS, which allows us to seek more efficient models and mixed precision quantization settings in a larger search space. These algorithm designs prioritize task-related performance, considering the performance-efficiency trade-off to some extent.

Algorithmic efficiency alone is insufficient to exploit the potential in the machine learning software stack. In the next chapter, we move to machine learning compilation, which has different design principles from machine learning algorithms.

Chapter 3

Efficient Machine Learning Compilation

Given a machine learning workload, the compiler transforms high-level computation graphs into low-level optimized executable instructions that can run efficiently on target hardware platforms. Aimed at maximizing the efficiency of completing machine learning workloads, it involves a series of steps, including graph analysis and simplification, code generation and optimization, system management, etc. The compilation bridges the gap between algorithms and real execution on the hardware platforms. As the complexity and scale of machine learning workloads continue to grow, there is a pressing need for efficient machine learning compilation that can optimize execution time and resource utilization.

This chapter is based on the following publications.

1. Zixuan Jiang, Keren Zhu, Mingjie Liu, Jiaqi Gu, David Z Pan. "An Efficient Training Framework for Reversible Neural Architectures". European Conference on Computer Vision (ECCV), 2020 [87].
2. Zixuan Jiang, Jiaqi Gu, Mingjie Liu, Keren Zhu, David Z Pan. "Optimizer Fusion: Efficient Training with Better Locality and Parallelism". Hardware Aware Efficient Training (HAET) workshop, International Conference on Learning Representations (ICLR), 2021 [84].
3. Zixuan Jiang, Jiaqi Gu, Hanqing Zhu, David Z. Pan. "Pre-RMSNorm and Pre-CRMSNorm Transformers: Equivalent and Efficient Pre-LN Transformers". This paper is under review at the Conference on Neural Information Processing Systems (NeurIPS), 2023. Its preprint version is public on arXiv (arXiv:2305.14858) [86].

I am the main contributor in charge of problem formulation, algorithm development, and experimental validations.

The compilation is closer to the target hardware compared to machine learning algorithms. For example, one of the critical challenges in machine learning compilation is leveraging the inherent parallelism and hardware-specific features of modern computing platforms. Efficient compilation techniques optimize for platform-specific characteristics to unlock the full potential of hardware accelerators, resulting in significant speedups and improved energy efficiency. Furthermore, these techniques consider the specific constraints and characteristics of the target hardware platforms, including memory hierarchies, storage capacities, and parallelism capabilities. The resulting executable code can fully utilize available hardware resources by tailoring the compilation process to exploit these platform-specific features, leading to more efficient computations.

In this chapter, we present three efficient compilation methods, which ensure arithmetic equivalence. In the first method, we remove the inherent redundancy in Pre-LN Transformers to obtain more efficient Pre-RMSNorm and Pre-CRMSNorm Transformers. These Transformer variants are arithmetically equivalent. Thus, as a free lunch, we may directly replace Pre-LN Transformers (e.g., GPT and ViT) with our proposed variants to achieve higher efficiency. The second one defines and solves the scheduling problem for reversible neural networks. The optimal scheduling enables us to achieve the fasted training throughput. We improve the efficiency of the computation and memory resource utilization. The third approach, optimizer fusion, accelerates the training process by considering the locality in memory hierarchy and parallelism in the algorithm. It is an enhancement of scheduling the computation graphs in eager execution.

3.1 Pre-RMSNorm and Pre-CRMSNorm Transformers: Equivalent and Efficient Pre-LN Transformers

Transformers have become a successful architecture for a wide range of machine learning applications, including natural language [183], computer vision [47], and reinforcement learning [29]. It is one of the foundation models [16], and pretrained Transformers [147] demonstrated impressive generalization results. Among the components of Transformers, normalization plays a critical role in accelerating and stabilizing the training process [117]. Layer Normalization (LayerNorm, LN) [5] and Root Mean Square Normalization (RMSNorm) [210] are two common normalization layers in Transformers. LayerNorm is in the original Transformer architecture [183], recentering and rescaling the input vector in $\mathbb{R}^d$ to obtain a zero-mean and unit-variance output. RMSNorm only rescales the input vector with its RMS value, offering greater computational efficiency than LayerNorm.

The machine learning community does not reach a consensus regarding the preferred normalization technique for Transformers. LayerNorm demonstrates remarkable success in the milestone Transformers, such as GPT [21,148] and ViT [47]. It is still the default normalization layer when building a new Transformer. On the contrary, RMSNorm is reported to accelerate the training and inference with similar performance as LayerNorm in Transformers. It has gained popularity in recent large language models, such as T5 [150], Gopher [149], Chinchilla [68], and LLaMA [181]. However, concerns persist regarding the potential negative impact of RMSNorm on the representation ability of Transformers. It remains an open question to determine the preferred normalization type for Transformers, requiring further theoretical and empirical investigation.

3.1.1 Overview

In this work, we aim to mitigate the discrepancy between LayerNorm and RMSNorm in Transformers. When delving into the prevalent Transformer architectures, Pre-LN and Pre-RMSNorm Transformers, we identify an opportunity to **unify** them by removing the inherent redundancy in Pre-LN models. In particular, the main branch vectors in the Pre-LN Transformers are always normalized before they are used, implying that the mean information is redundant. We can recenter the main branch without impact on the functionality of the models, which allows us to reduce LayerNorm to RMSNorm.

We further propose Compressed RMSNorm (CRMSNorm), which takes a vector in $\mathbb{R}^{d-1}$ as input, decompresses it to a zero-mean vector in $\mathbb{R}^d$, and applies the RMSNorm on the decompressed vector. Building upon this new normalization, we propose Pre-CRMSNorm Transformer that employs lossless compression on the zero-mean vectors. We apply such compression to zero-mean activations and parameters in Pre-RMSNorm Transformers, enhancing the efficiency of Pre-RMSNorm Transformers while preserving the equivalent arithmetic functionality.

We formally claim that Pre-LN, Pre-RMSNorm, and Pre-CRMSNorm Transformer variants are equivalent for both training and inference. Figure 3.1 visualizes the overview of their equivalence. Such equivalence directly enables more efficient training and deployment of Pre-LN Transformers. We can translate a Pre-LN model into an equivalent Pre-(C)RMSNorm model, which can be readily deployed or adapted to downstream tasks. The conversion process incurs minimal costs. We can also train a Pre-(C)RMSNorm model directly as if we train an equivalent Pre-LN Transformer counterpart.

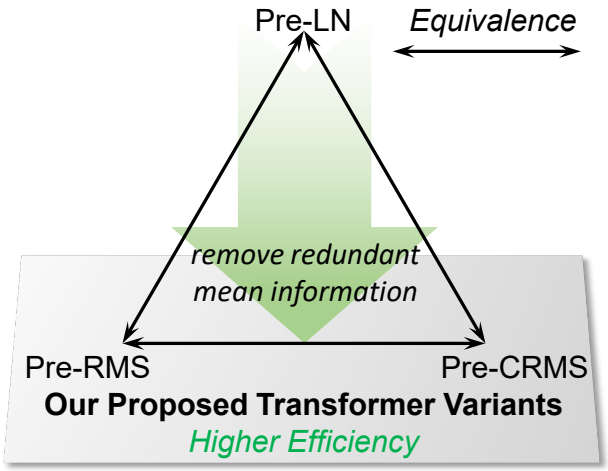


Figure 3.1: Overview of the three equivalent Transformer variants.

We highlight our contributions as follows.

- We achieve the **first-ever unification** of LayerNorm and RMSNorm in pre-normalization Transformers with proven arithmetic equivalence.
- We propose two variants: Pre-RMSNorm and Pre-CRMSNorm Transformers. The original Pre-LN Transformer and our proposed two variants are **equivalent** and can seamlessly interchange without affecting functionality.
- Our proposed architectures are 1%∼10% more **efficient** than the original Pre-LN Transformer for both training and inference. Such efficiency gains are effortlessly obtained without the need for fine-tuning or calibration. Our methods are orthogonal and complementary to most work improving Transformer efficiency.

Our implementation is available on GitHub. [1]

[1] https://github.com/zixuanjiang/pre-rmsnorm-transformer

3.1.2 Background

We introduce LayerNorm, RMSNorm, and their usage in Transformers. We provide an abstraction for Pre-LN Transformers.

3.1.2.1 LayerNorm and RMSNorm

Layer Normalization (LayerNorm, LN) [5] is a technique to normalize the activations of intermediate layers of neural networks. Given a vector $\boldsymbol{x} \in \mathbb{R}^d$, LayerNorm normalizes it to obtain a zero-mean unit-variance vector,

$$\text{LayerNorm}(\boldsymbol{x}) = \frac{\boldsymbol{x} - \mu(\boldsymbol{x})\mathbf{1}}{\sqrt{\|\boldsymbol{x}\|_2^2/d - \mu^2(\boldsymbol{x}) + \epsilon}}, \text{where } \mu(\boldsymbol{x}) = \frac{\mathbf{1}^T\boldsymbol{x}}{d}, \epsilon > 0. \tag{3.1}$$

LayerNorm recenters and rescales the activations and gradients in the forward and backward computations [202], which enables fast and robust training of neural networks.

Root Mean Square Normalization (RMSNorm) [210] is another technique used for normalizing the activations. It is similar to LayerNorm in that it aims to accelerate and stabilize the training but uses a different normalization approach. Instead of normalizing the inputs based on their mean and variance, RMSNorm normalizes them based on their root mean square (RMS) value. It is defined in the following equation,

$$\text{RMSNorm}(\boldsymbol{x}) = \frac{\boldsymbol{x}}{\sqrt{\|\boldsymbol{x}\|_2^2/d + \epsilon}}, \text{where } \epsilon > 0. \tag{3.2}$$

RMSNorm only rescales the input vector and the corresponding gradients, discarding the recentering process. As shown in their definitions, RMSNorm is computationally simpler and more efficient than LayerNorm. It is reported that replacing LayerNorm with RMSNorm can achieve comparable performance and save training and inference time by $7 - 64\%$ [210].

Given a zero-mean vector $\boldsymbol{x}$, these two kinds of normalization are equivalent. Formally, if $\mu(\boldsymbol{x}) = 0$, then LayerNorm($\boldsymbol{x}$) = RMSNorm($\boldsymbol{x}$). We may optionally introduce learnable parameters and apply an element-wise affine transformation on the output of LayerNorm and RMSNorm.

3.1.2.2 Normalization in Transformers

Normalization plays a crucial role and has many variants in Transformers [183]. LayerNorm is widely used in Transformer architectures to address this issue. The position of LN within the architecture is essential for the final performance. While the initial Transformer uses Post-LN, most Transformers employ Pre-LN to achieve more stable training, even though this can result in decreased performance [201]. Pre-LN is the mainstream normalization in Transformers, especially the large models, such as ViT [42, 47], PaLM [36], and GPT-series models [21, 148].

RMSNorm is proposed as an alternative normalization technique in Transformers. Several large language models, such as Chinchilla [68] and LLaMA [181], use Pre-RMSNorm in their blocks [217]. RMSNorm can help accelerate the training and inference with similar performance in these large models. Specifically, the experiments in [136] show that RMSNorm improves the pre-training speed by 5% compared with the LayerNorm baseline.

It is challenging to convert Transformers with one normalization to the other type. It is not clear which version of normalization is more suitable for Transformers.

3.1.2.3 Pre-LN Transformer

Figure 3.2.a illustrates the Pre-LN Transformer architecture, which consists of three parts, preprocessing, a stack of L blocks, and postprocessing.

Preprocessing. We preprocess the raw inputs, which range from paragraphs in natural language [183], images [47], to state-action-reward trajectories in reinforcement learning problems [29]. We import special tokens (such as the classification token) and embeddings (such as positional embeddings), ultimately obtaining a sequence of token embeddings $\boldsymbol{x}_0$.

Transformer blocks. The main body of Pre-LN Transformer [201] is a stack of residual blocks

$$\boldsymbol{x}_{l+1} = \boldsymbol{x}_l + \mathcal{F}_l(\boldsymbol{x}_l), \quad l = 0, 1, ..., L-1, \tag{3.3}$$

where $\boldsymbol{x}_l$ is the input of the l-th block, $\mathcal{F}_l$ is a sequence of operators *LayerNorm* $\rightarrow$ *Linear* $\rightarrow$ g_l $\rightarrow$ *Linear*. We name $\boldsymbol{x}_l$ as the vectors on the main branch and $\mathcal{F}_l$ as the residual branch [63]. The block $\mathcal{F}_l$ is usually an attention or a multi-layer perceptron (MLP) module. If g_l is an activation function, such as GELU [65], then the block $\mathcal{F}_l$ is a two-layer MLP. If g_l is a (masked) multi-head scaled dot product attention, then the block is the (casual) attention module [183]. These two linear layers are usually explicitly defined. Taking the attention module as an example, the input linear projection generates the query, key, and value vectors, while the output projection is applied to the concatenated results of all heads. If they are not explicitly defined, we can add an identity mapping, a special linear layer. Most of the learnable parameters of Transformers are in these two linear layers. [2]

[2]The elementwise affine transformation of LayerNorm or RMSNorm is also a linear transformation. Thus, it can be fused with the input linear projection. We disable this transformation in the normalization layers to simplify the analysis.

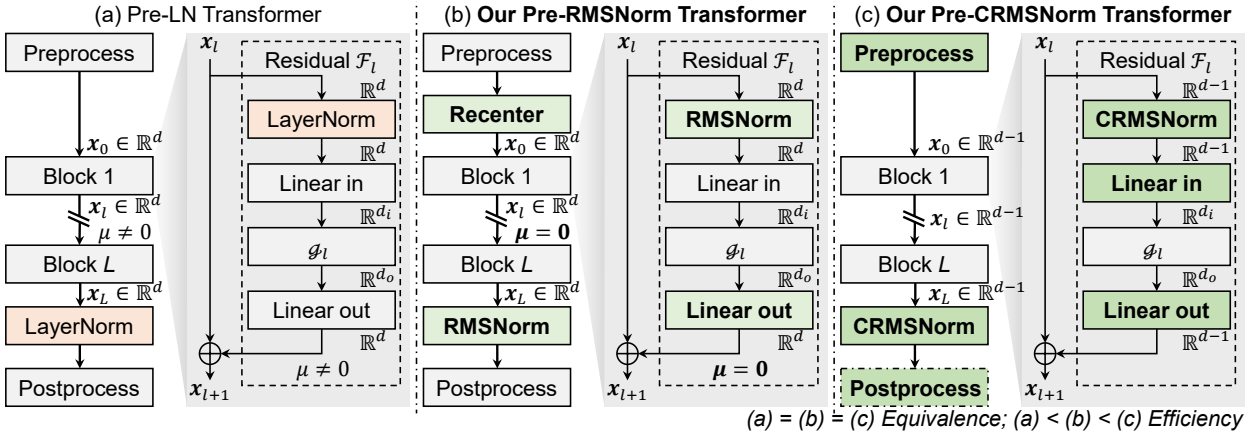


Figure 3.2: **Left.** The original Pre-LN Transformer architecture. **Middle and Right.** Our proposed Pre-RMSNorm and Pre-CRMSNorm Transformer architectures. These three architectures are equivalent. The differences are highlighted in bold and green blocks.

LayerNorm and postprocessing. We finally process the result $\mathrm{LN}(\boldsymbol{x}_L)$ to obtain the task-related results, such as classification probabilities. We apply the layer normalization on $\boldsymbol{x}_L$ since it usually has a high variance because it is the accumulation of all the Transformer blocks.

3.1.3 Method

We propose Pre-RMSNorm and Pre-CRMSNorm Transformer variants, shown in Figure 3.2, and claim that Pre-LN, Pre-RMSNorm, and Pre-CRMSNorm Transformers are arithmetically equivalent.

$$\texttt{Pre-LN Transformer} = \texttt{Pre-RMSNorm Transformer} = \texttt{Pre-CRMSNorm Transformer}. \tag{3.4}$$

We will show the equivalence of these three architectures and then analyze the computational efficiency improvement by our proposed model variants. We discuss the Post-LN in Appendix

B.2.

3.1.3.1 Pre-LN Transformer = Pre-RMSNorm Transformer

LayerNorm is invariant to the shifting $\text{LN}(\boldsymbol{x} + k\mathbf{1}) = \text{LN}(\boldsymbol{x}), \forall k \in \mathbb{R}$. We observe that LayerNorm is applied to the main branch vectors before they are used in either residual branches or the final postprocessing. Therefore, we can replace the main branch vectors $\boldsymbol{x}_l$ with $\boldsymbol{x}_l + k_l\mathbf{1}, \forall k_l \in \mathbb{R}$ without impact on the functionality of the Pre-LN Transformer. If $k_l = -\mu(\boldsymbol{x}_l)$, we replace the main branch vectors with its recentered version $\boldsymbol{x}_l - \mu(\boldsymbol{x}_l)\mathbf{1}$. We can explicitly maintain zero-mean main branches with the same arithmetic functionality.

We propose three modifications to the original Pre-LN Transformer to obtain an equivalent Pre-RMSNorm Transformer.

1. Recenter the $\boldsymbol{x}_0$ before the first Transformer block, where $\text{Recenter}(\boldsymbol{x}) = \boldsymbol{x} - \mu(\boldsymbol{x})\mathbf{1}$.
2. For the output projection in residual branches, replace the weight $\boldsymbol{A}_o$ and bias $\boldsymbol{b}_o$ with $\hat{\boldsymbol{A}}_o = \boldsymbol{A}_o - \frac{1}{d}\mathbf{1}\mathbf{1}^T\boldsymbol{A}_o, \hat{\boldsymbol{b}_o} = \boldsymbol{b}_o - \mu(\boldsymbol{b}_o)\mathbf{1}_o$, where d is the dimension of $\boldsymbol{x}_0$.
3. Replace LayerNorm with RMSNorm at the beginning of residual blocks and before postprocessing.

Since $\mu(\boldsymbol{x}_{l+1}) = \mu(\boldsymbol{x}_l) + \mu(\mathcal{F}_l(\boldsymbol{x}_l))$, we can keep zero-mean on the main branch *if and only if* the input of the first block $\boldsymbol{x}_0$ and the output of each residual branch $\mathcal{F}_l$ are re-centered with zero-mean. The first modification is to recenter $\boldsymbol{x}_0$, while the second modification is to recenter the output of residual branches. For the residual branch $\mathcal{F}_l$, the ending linear transformation enables us to recenter its output without extra computation, implied by Lemma 3.1.1. We can recenter the weight and bias of a linear layer to recenter its output.

Lemma 3.1.1 *Given a linear transformation* $\boldsymbol{y} = \boldsymbol{A}\boldsymbol{x} + \boldsymbol{b}, \boldsymbol{x} \in \mathbb{R}^n, \boldsymbol{A} \in \mathbb{R}^{m \times n}, \boldsymbol{b}, \boldsymbol{y} \in \mathbb{R}^m$*, we can decompose the output with two parts* $\boldsymbol{y} = \hat{\boldsymbol{A}}\boldsymbol{x} + \hat{\boldsymbol{b}} + \mu(\boldsymbol{y})\mathbf{1}$*. The first part* $\hat{\boldsymbol{A}}\boldsymbol{x} + \hat{\boldsymbol{b}} = \boldsymbol{y} - \mu(\boldsymbol{y})\mathbf{1}$*, with zero mean, is another linear transformation with* $\hat{\boldsymbol{A}} = \boldsymbol{A} - \frac{1}{m}\mathbf{1}\mathbf{1}^T\boldsymbol{A}, \hat{\boldsymbol{b}} = \boldsymbol{b} - \mu(\boldsymbol{b})\mathbf{1}$.

The first two modifications ensure that we maintain zero mean on the main branch vectors. Given a zero-mean input, LayerNorm is equivalent to RMSNorm, which implies that the third modification has no impact on the functionality. With these modifications, we demonstrate that Pre-LN and Pre-RMSNorm Transformers are equivalent.

3.1.3.2 Pre-RMSNorm Transformer = Pre-CRMSNorm Transformer

For a zero-mean vector $\boldsymbol{x} \in \mathbb{R}^d$, we can compress it losslessly by discarding its last element. In the decompression, we recover the discarded element with $x_d = -\sum_{i=0}^{d-1} x_i$. The decompression has an extra cost, while the compression does not induce extra computation. The space-saving ratio of this compression method is $1/d$.

We define **Compressed Root Mean Square Normalization** (CRMSNorm), which takes a vector $\boldsymbol{x} \in \mathbb{R}^{d-1}$ as input. CRMSNorm first decompresses the vector $\boldsymbol{x}$ to obtain a zero-mean vector in $\mathbb{R}^d$, then applies RMSNorm on the zero-mean vector. It can generate the normalized zero-mean results in either $\mathbb{R}^{d-1}$ or $\mathbb{R}^d$. Its formal definition is in Equation 3.5.

$$\text{CRMSNorm}(\boldsymbol{x}) = \frac{\boldsymbol{x}}{\sqrt{(\sum_{i=1}^{d-1} x_i^2 + (\sum_{i=1}^{d-1} x_i)^2)/d + \epsilon}}, \text{ where } \boldsymbol{x} \in \mathbb{R}^{d-1}. \tag{3.5}$$

We simplify the Pre-RMSNorm Transformers to obtain the Pre-CRMSNorm Transformers with the following modifications.

1. Compress the zero-mean main-branch vectors from $\mathbb{R}^d$ to $\mathbb{R}^{d-1}$. Preprocessing and postprocessing handle compressed vectors in $\mathbb{R}^{d-1}$.

2. Replace RMSNorm with CRMSNorm.

3. Simplify the weight $\boldsymbol{A}_i$ in the input projection layer. Let $\boldsymbol{a}_d$ be the last column vector of $\boldsymbol{A}_i$. $\hat{\boldsymbol{A}}_i = \boldsymbol{A}_i - \boldsymbol{a}_d \mathbf{1}^T$ is the compressed weight matrix.

4. Simplify the weight and bias in the output projection layer. We discard the last row of the weight matrix $\hat{\boldsymbol{A}}_o$ and the last element of the bias $\hat{\boldsymbol{b}_o}$ since they are used to generate the redundant last element.

We compress the zero-mean activations in our proposed Pre-RMSNorm Transformers and correspondingly simplify the two linear layers in residual branches.

We can fuse preprocessing, recentering, and compression. The fused preprocessing generates compressed vectors in $\mathbb{R}^{d-1}$ to represent zero-mean vectors in $\mathbb{R}^d$. Taking language models as an example, we can recenter and compress (discard the last element) the word and position embeddings.

For the input linear projection, its input is the output of RMSNorm, whose mean is zero. We compress the output of the RMSNorm and the weight of the linear layer. Specifically, if the linear layer takes a zero-mean vector as input, then

$$\text{Linear}(\boldsymbol{x}) = \boldsymbol{A}_i \boldsymbol{x} + \boldsymbol{b}_i = (\boldsymbol{A}_i - \boldsymbol{a}_d \mathbf{1}^T)\boldsymbol{x} + \boldsymbol{b}. \tag{3.6}$$

$\hat{\boldsymbol{A}}_i = \boldsymbol{A}_i - \boldsymbol{a}_d \mathbf{1}^T$ is the compressed weight matrix since its last column is a zero vector. The simplified linear layer only needs the compressed zero-mean vectors in $\mathbb{R}^{d-1}$ as input.

For the output linear projection, we only need to calculate the first $(d-1)$ elements for the vectors in $\mathbb{R}^d$. Thus, we can compress the weight $\hat{\boldsymbol{A}}_o$ and bias $\hat{\boldsymbol{b}_o}$. As shown in Figure 3.2, the shape of $\hat{\boldsymbol{A}}_o$ is (d, d_o), we can compress it directly to $(d-1, d_o)$ by discarding its last row. Similarly, we can compress $\hat{\boldsymbol{b}_o} \in \mathbb{R}^d$ by discarding its last element. This weight compression also reflects their redundancy. As shown in Section 3.1.3.1, $\hat{\boldsymbol{b}_o}$ and all column vectors of $\hat{\boldsymbol{A}}_o$ are zero-mean.

Figure 3.2 demonstrates the difference in vector dimension. We demonstrate the equivalence between Pre-RMSNorm and Pre-CRMSNorm Transformers.

3.1.3.3 Pre-LN Transformer = Pre-CRMSNorm Transformer

To translate a pre-trained Pre-LN Transformer to a Pre-CRMSNorm Transformer, we can convert it into a Pre-RMSNorm model and then finish the conversion. For the model implementation, we need two steps to convert a Pre-LN Transformer to Pre-CRMSNorm Transformer.

1. Reduce the hidden dimension from d to $d-1$. [3]
2. Replace LayerNorm with CRMSNorm.

Namely, Pre-LN Transformers with the main branch vectors in $\mathbb{R}^d$ are equivalent to Pre-CRMSNorm Transformers in $\mathbb{R}^{d-1}$, which further echoes the redundancy in the Pre-LN Transformers.

[3] We reduce the hidden dimension in the preprocessing, main branch, two linear layers in residual branches, and postprocessing. We keep the d_i, d_o in residual branches. For instance, we maintain the MLP dimension and head dimension (dimension of query, key, and value vectors).

3.1.3.4 Training and Inference Efficiency

We show how to make conversions between the three variants. The conversions only consist of one-time parameter adjustments without expensive fine-tuning or calibration, similar to operator fusion [140]. We qualitatively analyze the efficiency improvement of our proposed Pre-(C)RMSNorm Transformers compared with equivalent Pre-LN models.

Pre-RMSNorm Transformer We discuss the impact of three modifications on training and inference, as shown in Table 3.1.

	Recenter	Zero-Mean Linear	RMSNorm
Training	a little increase	a little increase	decrease
Inference	same	same	

Table 3.1: The computation workload of our Pre-RMSNorm model compared with the original Pre-LN Transformer.

The linear layer with zero-mean output. The modified linear layer will not induce extra computation for inference since we can replace the weight and bias in advance. During inference, we can treat it as a standard linear layer with equivalently transformed parameters.

We have to pay the extra cost for training for the parameter change. The parameter optimizer manages $\boldsymbol{A}_o, \boldsymbol{b}_o$, but we use their recentered version $\hat{\boldsymbol{A}}_o, \hat{\boldsymbol{b}_o}$ during training. The induced cost is small for several reasons. (1) The size of parameters $\boldsymbol{A}_o, \boldsymbol{b}_o$ is relatively small since they do not depend on the batch size and sequence length. For reference, the computation cost of LayerNorm in the original Pre-LN Transformer is proportional to the batch size and the sequence length. (2) Obtaining $\hat{\boldsymbol{A}}_o, \hat{\boldsymbol{b}_o}$ can be done ahead of time since

it does not depend on the input. We may leverage the idle time of accelerators to compute them. (3) In data-parallel distributed training, the parameter server [110] manages the parameters. Each worker will receive $\hat{\boldsymbol{A}}_o, \hat{\boldsymbol{b}_o}$ from the server and then pass the gradients of loss to $\hat{\boldsymbol{A}}_o, \hat{\boldsymbol{b}_o}$ to the server. Only the server needs to maintain and update the original $\boldsymbol{A}_o, \boldsymbol{b}_o$. In short, it is much easier to process the model parameters than the intermediate activations.

Recentering. It is possible to fuse the recentering with the preprocessing, which usually handles the sum of several kinds of embeddings. For example, the input embeddings of the BERT model [45] are the accumulation of the token embeddings, the segmentation embeddings, and the position embeddings. Since $\text{Recenter}(\boldsymbol{x}+\boldsymbol{y}) = \text{Recenter}(\boldsymbol{x})+\text{Recenter}(\boldsymbol{y})$, recentering the input is equivalent to recentering each embedding before the addition. We can recenter the accumulated embeddings or each embedding separately before the accumulation. Suppose an embedding is from a linear layer. In that case, we can modify the linear layer such that it generates the zero-mean output, similar to how we edit the output linear projection.

For inference, we can recenter the related embeddings or linear layers in advance such that no extra computation is induced. For training, recentering induces extra cost since it is on the fly.

Replacing LayerNorm with RMSNorm. Section 3.1.2.1 introduces that RMSNorm can achieve speedup compared with LayerNorm, as demonstrated by the previous models. This replacement can help us accelerate the training and inference of the Pre-LN Transformer.

Pre-CRMSNorm Transformer CRMSNorm, an extension of RMSNorm, saves execution time as it is more computationally efficient than LayerNorm. Additionally, CRMSNorm further reduces the hidden dimension from d to $d-1$, which can reduce the model size, computation, communication, and memory consumption by $1/d$ in theory. However, most accelerators can not efficiently handle the vectors in $\mathbb{R}^{d-1}$ when d is a large even number, especially a power of two (e.g., 1024, 4096). This limitation arises because these accelerators are typically optimized for arithmetic with even dimensions. In some cases, handling $\mathbb{R}^{d-1}$ vectors may take much more time than $\mathbb{R}^d$ vectors. Thus, we need to examine if the accelerators can support $\mathbb{R}^{d-1}$ vectors efficiently. If not, we have the following alternatives. For inference, we may either (1) add zero embeddings, or (2) decompress the vectors and translate the model into the Pre-RMSNorm variant. For training, we suggest keeping the hidden dimension d, which is equivalent to a Pre-LN model with the hidden dimension $d+1$. In this way, the CRMSNorm can help us increase the model representability and computation efficiency at the same time.

3.1.4 Experiments

We claim that our major contributions are the unification and equivalence of the three Transformer variants. The efficiency is a free lunch accompanying the equivalence. Also, the efficiency of RMSNorm over LayerNorm has been shown in the previous work [136, 210]. We focus on analyzing the efficiency of each component of our method in this section.

We conduct experiments on ViT [47, 179] and GPT3 [21] since they represent two mainstream architectures of Transformers, encoder-only and casual decoder. Other Transformer variants can be treated as an extension of these two architectures, such as encoder-

decoder [183], and non-causal decoder [209]. Also, ViT and GPT cover the areas of computer vision and natural language, where Transformers have been popular and achieved great success.

We use PyTorch 2.0 [142] to build the training and inference pipeline. We run iterations at least 100 times and report the 25th, 50th (median), and 75th percentile since these quartiles are more robust than the mean. We use the automatic mixed precision [133] for both training and inference. We provide more results for different machine learning frameworks (JAX [19]), precision and datatype, and computation platforms. Please see Appendix B.3 for more details.

3.1.4.1 Experiments on ViT

The ViT takes images with 3 channels and a resolution of 224×224 and generates a classification over 1000 classes, which is the standard setting for ImageNet training and inference.

In the training recipe of ViT [47], dropout [170] is added at the end of each residual branch, which breaks the zero-mean property of the residual output. Hence, in Pre-RMSNorm Transformers, we need to recenter the output vectors explicitly at the end of the residual branch. [4] The Pre-CRMSNorm Transformers are compatible with dropout since it uses vectors in $\mathbb{R}^{d-1}$ to represent the compressed zero-mean vectors in $\mathbb{R}^d$. On the contrary, DeiT [179] disables the dropout and can guarantee the zero-mean output, in spite that the stochastic depth [72] and LayerScale [180] are applied. We follow the settings in DeiT [179]

[4] Dropout only impacts training and has no impact on inference. Thus, the Pre-RMSNorm variant can also be used for inference.

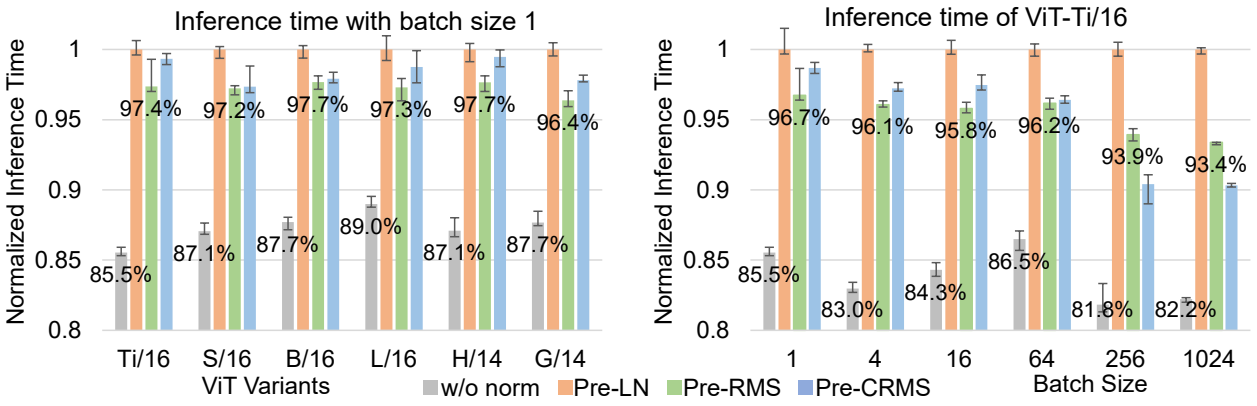


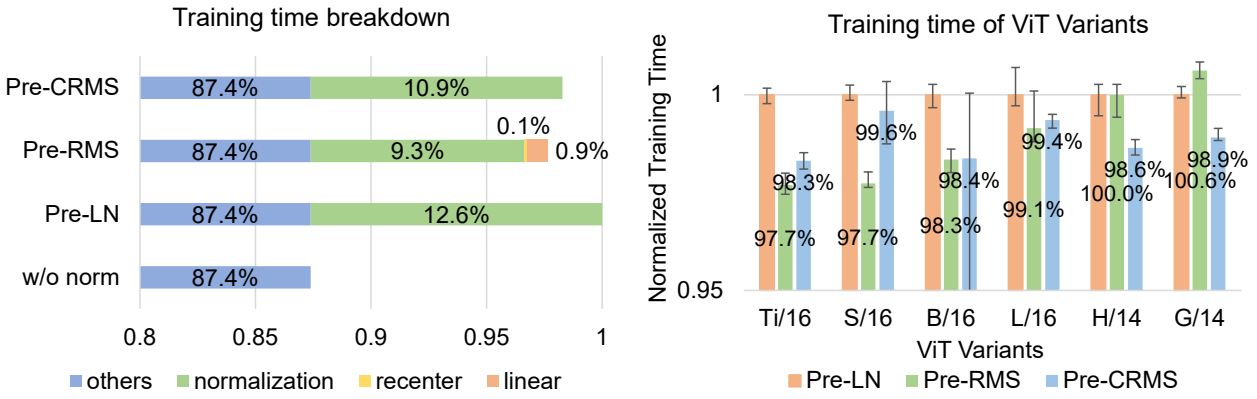
Figure 3.3: Normalized inference time on ViT with different model sizes and batch sizes.

in this section.

Inference. Figure 3.3 illustrates the inference time with different batch sizes and model sizes on a single A100 GPU. Taking the Pre-LN Transformer as the baseline, our proposed Pre-RMSNorm Transformer can reduce the inference time by $1\% - 9\%$. This reduction takes effect for ViTs with different models and various mini-batch sizes. The left subfigure demonstrates the inference latency, which is the inference time when the batch size is 1.

The proportion of LayerNorm in the total computation is $12\% - 18\%$ in these experiments. As discussed in Sections 3.1.2.1 and 3.1.3.4, replacing LayerNorm with RMSNorm can help us accelerate the inference. RMSNorm can reduce the inference time of LayerNorm by $20\% - 60\%$, thus helping us achieves a stale faster inference for Pre-RMSNorm models.

For Pre-CRMSNorm, the GPU cannot efficiently handle vectors in $\mathbb{R}^{d-1}$ in some cases. We add zero padding for these cases to obtain vectors in $\mathbb{R}^d$. With zero padding, Pre-CRMSNorm models are less efficient than Pre-RMSNorm ones due to the extra decom-



a Training time breakdown on ViT-Ti/16.

b Normalized training time

Figure 3.4: Training time comparison and breakdown on ViT variants

pression. However, for the cases where $\mathbb{R}^{d-1}$ vectors can be accelerated efficiently, we can achieve an even 10% time reduction.

We also conduct experiments (1) with other precisions, (2) on CPUs, (3) with JAX [19] and observe similar performance. For these ViT variants, we have achieved an average of 3.0% inference time reduction.

Training. We train ViTs on a single workstation with 4 A100s with data parallel training [111], following the DeiT training recipe. Each training iteration consists of forward and backward computation on all the workers, gradient all-reduce, parameters update with an optimizer, and broadcasting the new parameters. Figure 3.4a visualizes the breakdown of the related components. In the Pre-LN Transformer, the layer normalization accounts for 12.6% of total training time. For the Pre-RMSNorm variant, we have to modify the output projection and recenter the input, which induces the 0.89% and 0.09% extra computation time. Then LayerNorm is reduced to RMSNorm, whose computation cost is 9.27%. Overall, the Pre-RMSNorm reduces the training time by 2.36%.

We train the Pre-CRMSNorm Transformers with d as the hidden dimension since we do not obtain a speedup from the compressed dimension since the GPUs cannot handle $\mathbb{R}^{d-1}$ vectors efficiently. The CRMSNorm is more computationally expensive than the RMSNorm given the same input but takes less time than LayerNorm. The Pre-CRMSNorm Transformer does not need the recentering and special linear layers to generate zero-mean results at the end of residual branches. Above all, training the Pre-CRMSNorm variant is 1.74% faster than the Pre-LN model.

Figure 3.4b illustrates the training time of different ViTs. We have achieved a speedup of $1\% - 2.5\%$, which is smaller than the inference speedup. Considering only the forward and backward computation, the speedup is similar between training and inference. Nevertheless, the training needs extra time on gradient all-reduce, optimizer update, and parameters broadcast, which shrinks the percentage of normalizations in the whole computation. For reference, the percentage is $10 - 15\%$ for training these ViTs.

3.1.4.2 Experiments on GPT

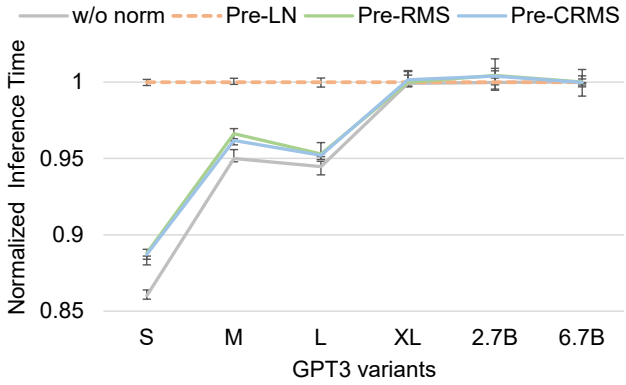


Figure 3.5: GPT3 inference performance

We measure the inference time on the GPT3 [21] variants with batch size 1 and sequence length 512. The results are shown in Figure 3.5. In small GPT3 models, layer normalization takes considerable time. Hence, replacing the LayerNorm with (C)RMSNorm can reduce the inference time by up to 10%. However, as the model grows, the attention and MLP take charge of the main part of the computation [109]. The normalization takes $< 1\%$ of the inference time for GPT3 XL and larger models, which is the upper bound of the speedup with our methods.

We discuss the reason that the normalization takes less percentage of the computation. Let L be the input sequence length and d be the dimension of embedding vectors. The computation complexity of either LayerNorm or RMSNorm is $\mathcal{O}(Ld)$, which is smaller than the attention and MLP modules, whose complexities are $\mathcal{O}(L^2d)$ and $\mathcal{O}(Ld^2)$. As L and d increase, the attention and MLP have a larger portion of the total computation.

Applying quantization can mitigate this issue. By applying int8 matrix multiplication [44], the percentage of the layer normalization increases to 10% for GPT3 XL and 2.7B. Therefore, our Pre-RMSNorm can reduce the inference time by 4% for them.

Training performance is similar to the ViT one. We have achieved 1.5% and 1.8% time reduction for GPT3 Small and Medium, respectively.

3.1.4.3 Discussions

Although our relative improvement in the training and inference efficiency seems not large (up to 10% time reduction), we have the following arguments to support its significance.

Our proposed method can guarantee arithmetic equivalence and is a free lunch for Pre-LN Transformers. The efficiency improvement originates from removing the inherent

redundancy in Pre-LN Transformers without introducing any fine-tuning or calibration. Our work can strictly push the performance-efficiency Pareto frontier of Pre-LN Transformers.

The modest relative improvement can be translated into significant absolute improvement given that Pre-LN Transformers are foundation models for the current and future data-centric [208] and generative [25] artificial intelligence. For instance, reducing the ChatGPT inference cost by 1% may save $\$7,000$ per day [134].

Our method is orthogonal and complementary to most work improving efficiency, such as efficient Transformer variants [177], quantization [17, 118], and distillation to smaller models [176].

3.1.5 Summary

We propose two equivalent and efficient variants for the widely used Pre-LN Transformers. We point out the inherent redundancy in the Pre-LN Transformer. By maintaining zero-mean on the main branch vectors and thus simplifying LayerNorm, we obtain the Pre-RMSNorm architecture. We further apply a lossless compression on the zero-mean vectors to obtain the Pre-CRMSNorm model. For the first time, We unify these normalization variants within the Transformer model.

We enable the more efficient utilization of Pre-LN Transformers, allowing for seamless transitions between normalization techniques with minimal overhead. We can replace a Pre-LN Transformer with an equivalent Pre-(C)RMSNorm Transformer with better training and inference efficiency, which is a free lunch. We strictly push the performance-efficiency Pareto frontier of foundational Pre-LN Transformers. As a result, pre-trained Pre-LN Transformers (such as ViT and GPT) can be deployed more efficiently, and new equivalent or superior

models can be trained with higher throughput.

Extensions. We believe that our proposed CRMSNorm technique has the potential for application in other neural architectures. Further exploration of its usage in different contexts would be beneficial. Additionally, while our focus has been on pre-normalization Transformers, it would be valuable to investigate the application of our methods to other related architectures, especially the foundation models. Finally, integrating our proposed method into machine learning compilers could directly enable the generation of equivalent and simplified computation graphs.

Limitations. Detailed implementations and specific optimizations will play a crucial role in achieving practical performance gains. It is necessary to focus on developing highly optimized implementations of (C)RMSNorm to bridge the gap between theoretical and practical efficiency improvements. By addressing these challenges, we can fully leverage the potential benefits of (C)RMSNorm and enable more efficient utilization of Pre-LN Transformers.

3.2 An Efficient Training Framework for Reversible Neural Architectures

The backpropagation [157] mechanism is widely used in training neural networks. However, since intermediate results need to be saved for backward computations, the backpropagation requires considerable memory footprint. As neural networks become larger and deeper, the increasing memory footprint is forcing the usage of smaller mini-batch sizes. In extreme cases, deep networks have to be trained with a mini-batch size of 1 [220]. The issue of memory consumption impedes the explorations of desirable deep learning models,

especially in the field of large language models.

Researchers have proposed several methods to address the challenge of inflating memory footprint [168]. Chen et al. [32] propose the gradient checkpoint mechanism to store partial intermediate results. The discarded activations will be recovered through recomputations in the backward pass. The memory swapping method [156, 212] moves intermediate activations to other devices to reduce the memory footprint of the current device. The extra memory transfer imposes overhead on training efficiency. Reversible operators [56] allow recovering the intermediate feature maps in backward pass through the corresponding inverse functions. All these three methods reduce the memory footprint at the cost of extra computation or memory transfer. They do not affect the model accuracy as the training process is numerically unchanged.

Specifically, reversible neural architectures have been successfully adopted in computer vision research, e.g., the reversible U-net for volumetric image segmentation [22], and the reversible architecture for 3D high-resolution medical image processing [14]. The lower memory footprint allows deeper models to be trained, inducing more predictive capability and higher accuracy.

Figure 3.6 shows two extremes in neural network training. Standard backpropagation achieves the extreme of computation efficiency at the expense of the highest memory footprint, such that it does not contain any redundant computations. On the other extreme, the fully reversible strategy has the lowest memory footprint with imposing the greatest computation overhead. However, The design space between the two extremes is less studied. Existing research regarding reversible neural networks mainly focuses on saving memory consumption. All the reversible layers are executed in the memory-efficient mode. The

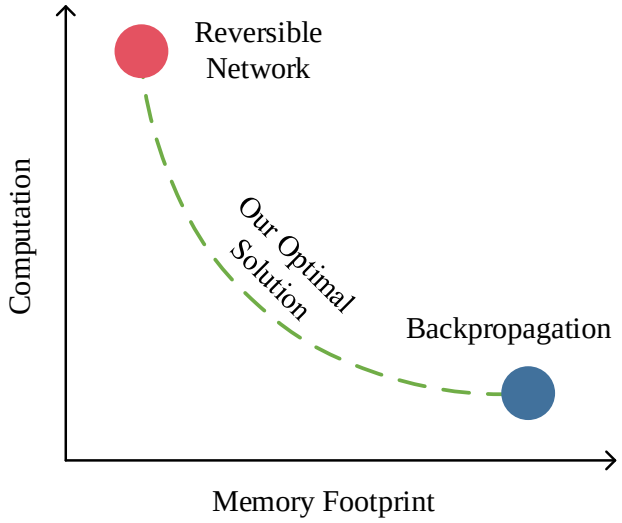


Figure 3.6: Two extremes when training neural networks. The lower right extreme stands for the standard backpropagation method, which does not contain any redundant computations. The upper left extreme can achieve the lowest memory footprint by fully leveraging the reversibility of the neural network.

computation overhead of their inverse functions is overlooked. In other words, most implementations of reversible neural networks do not explore the intermediate design space by considering the trade-off between computation cost and memory footprint.

3.2.1 Overview

We explore the design space by considering the trade-off between computation and memory footprint. We derive the mathematical formulation of the decision problem for reversible neural architectures. We formulate the training time as the objective function with memory usage as an optimization constraint. By showing that it is a standard 0/1 knapsack problem in essence, we use a dynamic programming algorithm to find the optimal solution. We also discuss the relationship between mini-batch size and training throughput.

Our contributions are highlighted as follows.

- **New Perspective.** We explore the design space for reversible neural architectures from a novel perspective of joint optimization.
- **Optimality.** Our framework guarantees to obtain the maximum training throughput for reversible neural architectures under given memory constraints.
- **Automation.** Our framework provides a fully automated solution, enabling more efficient development and training for reversible neural networks.

3.2.2 Background

In this section, we discuss the background of reversible neural architectures and the scheduling framework for the training process.

3.2.2.1 Reversible Neural Architectures

Figure 3.7a demonstrates a conventional non-reversible neural architecture. The layer $y = f(x)$ is non-reversible if and only if there is no inverse computation $x = f^{-1}(y)$ for the original function f. For a non-reversible layer, we often need to store its original input x during forward computation so that we can compute gradients during backpropagation. As an example, for a linear layer $y = f(x) = \theta^T x$, where θ represents the weight vector, its backward computation $\partial y/\partial\theta = x$ depends on the original input x.

Traditional neural networks are mostly based on these non-reversible layers. The memory consumed by the feature maps dominates the total memory utilization, especially

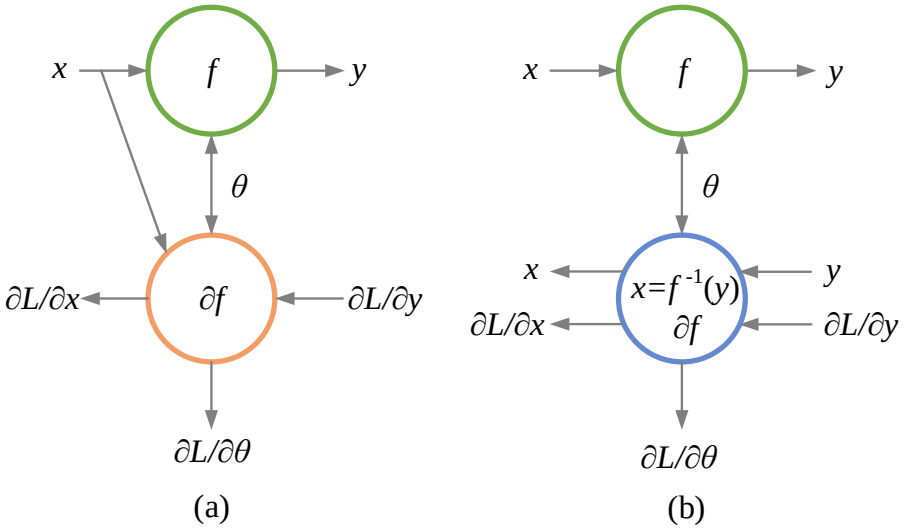


Figure 3.7: (a) non-reversible and (b) reversible neural architectures. For a non-reversible layer, we often need to save its original input x for backward computations. For a reversible layer, the original input x can be calculated via its inverse function $x = f^{-1}(y)$.

in deep neural networks [156]. Therefore, the memory footprint can decrease significantly by discarding those feature maps.

Figure 3.7b illustrates a reversible operator. When using the reversible layer $y = f(x)$, it is possible to recover x in the backward computation by calling its inverse function $x = f^{-1}(y)$. Therefore the memory consumption can be saved by discarding the intermediate feature map x.

Some of the commonly used operators in neural networks are implicitly reversible, such as convolution layers with a stride of 1 [93], and fully connected layers with invertible weight matrix. Inplace Activated Batch Normalization (ABN) [23] leverages the reversibility of the batch normalization [76] and some activation functions (such as leaky ReLU). Neural ordinary differential equations [30] can achieve constant memory usage through reversibility

in backpropagation.

Researchers also propose many variations of explicit reversible neural architectures [78]. The reversible residual architecture [56] does computations on a pair of inputs (x_1, x_2) as shown in Equation 3.7.

$$y_1 = x_1 + F(x_2), y_2 = x_2 + G(y_1) \tag{3.7}$$

It is reversible since the inputs can be recovered from output pairs as demonstrated in Equation 3.8.

$$x_2 = y_2 - G(y_1), x_1 = y_1 - F(x_2) \tag{3.8}$$

This technique can be combined with traditional recurrent neural networks to get reversible RNNs [125]. Kitaev et al. apply the above architecture to the Transformer [183] and obtain Reformer [95] as shown in Equations 3.9 and 3.10.

$$y_1 = x_1 + \text{Attention}(x_2), y_2 = x_2 + \text{FeedForward}(y_1) \tag{3.9}$$

$$x_2 = y_2 - \text{FeedForward}(y_1), x_1 = y_1 - \text{Attention}(x_2) \tag{3.10}$$

Moreover, reversible vision Transformers [127] decouples the memory requirement from the depth of the model. RevBiFPN [35] is a fully reversible bidirectional feature pyramid network.

Although the computation overhead is considered and discussed, these prior studies mainly focus on memory footprint reduction. They do not explore the space between the two extremes illustrated in Figure 3.6.

3.2.2.2 Scheduling for Training

For most developers, the primary concern regarding the training process is how to maximize the training throughput given existing machines, especially GPUs. Specifically, there is a need for a framework to automate the training process to fully utilize the computation capability and memory capacity of specific machines.

Frameworks for the scheduling problem with gradient checkpoints are great examples. The scheduling problem seeks the minimum computation overhead with a memory footprint constraint. Researchers propose many algorithms to find optimal solutions for gradient checkpoint selection. Kusumoto *et al*. provide a dynamic programming algorithm from the perspective of computation graphs [101]. Jain *et al*. formulate the scheduling problem as a mixed integer linear program and solve it via standard solvers [79]. However, a similar problem for reversible neural architectures does not get much attention. We formulate and solve this problem in this work.

There are also work focused on the scheduling for distributed training. Jia *et al*. optimize how each layer is parallelized in distributed and parallel training [81]. However, they do not consider the reversibility in operators. Our framework can be used directly in every single machine in the distributed training scenario.

3.2.3 Method

In this section, we first describe two modes for reversible neural architectures. We denote them `M-Mode` and `C-Mode`, respectively. We then formulate the decision problem, and propose an algorithm and our framework. We also discuss the problem when mini-batch sizes are not fixed.

mode	forward	backward	computation cost	memory cost
`M-Mode`	discard x	recover x from y	$x = f^{-1}(y)$	0
`C-Mode`	save x	use x directly	0	size of x

Table 3.2: Comparisons of two modes.

3.2.3.1 Memory Centric and Computation Centric Modes

Each reversible layer $y = f(x)$ can be computed in two modes during the training process. First, we can leverage its reversibility. We denote it `M-Mode`, which represents **m**emory centric **mode**. Precisely, we discard the activation x in forward computations, then recover it in the backward pass. This mode saves the memory consumed by x at the cost of inverse computation of $x = f^{-1}(y)$. Another mode is treating the reversible layer as a conventional non-reversible layer, which is denoted `C-Mode` representing **c**omputation centric **mode**. In this mode, we save the feature map x in the forward pass, then use it directly in the backward computation. This mode does not involve redundant computations but requires an extra memory footprint. Table 3.2 summarizes these two modes.

3.2.3.2 Formulation

Let f be a neural network with $(k+n)$ layers, among which there are n reversible layers $\{f_i\}_{i=1}^{n}$. For each of these n reversible layers, we can decide to do forward and backward computation following one of the modes above. Let $x \in \{0,1\}^n$ be the decision variable. $x_i = 0(1)$ means that the reversible layer f_i follows the `M-Mode` (`C-Mode`). Thus, for n reversible layers, the 2^n choices constitute the whole solution space.

The two extremes in Figure 3.6 can be written as $x = \mathbf{0}$ and $x = \mathbf{1}$. $x = \mathbf{0}$ represents

that we discard all the intermediate results to achieve the lowest memory footprint. We treat it as `baseline-M`. We denote the other extreme without redundant computations ($x = \mathbf{1}$) as `baseline-C`. Currently, most of the implementations of reversible neural networks use `baseline-M` directly.

Let $t_{f1}, t_{b1}(t_{f2}, t_{b2}) \in \mathbb{R}^n_{++}$ be the execution time vector of forward and backward pass in the `M-Mode` (`C-Mode`) respectively. Compared with the `C-Mode`, the extra execution time consumed by the `M-Mode` is $t_e = (t_{f1} + t_{b1}) - (t_{f2} + t_{b2})$. The total execution time of forward and backward computation of all these reversible layers $\{f_i\}_{i=1}^n$ are

$$(\mathbf{1} - x)^T(t_{f1} + t_{b1}) + x^T(t_{f2} + t_{b2}) = \mathbf{1}^T(t_{f1} + t_{b1}) - t_e^T x$$

Similarly, let $m \in \mathbb{Z}^n_{++}$ be the extra memory footprint of `C-Mode` compared with `M-Mode`, i.e., the size of corresponding intermediate activations. The total extra memory footprint of these feature maps is $m^T x$.

Finally, the time centric optimization problem can be written as Problem 3.11.

$$\begin{aligned} \min_x \quad & \mathbf{1}^T(t_{f1} + t_{b1}) - t_e^T x \\ \text{s.t.} \quad & m^T x + m_o \leq M \\ & x_i \in \{0, 1\}, i = 1, ..., n \end{aligned} \tag{3.11}$$

where M is the memory capacity of the machine, m_o represents the memory allocated for other tensors (such as feature maps of non-reversible layers, and neural network parameters) when we achieve peak memory in a training iteration. Users can also specify the memory capacity M explicitly.

For other parts of the training process, such as the optimizer, the computation of non-reversible layers, their execution time is constant and independent of our decisions.

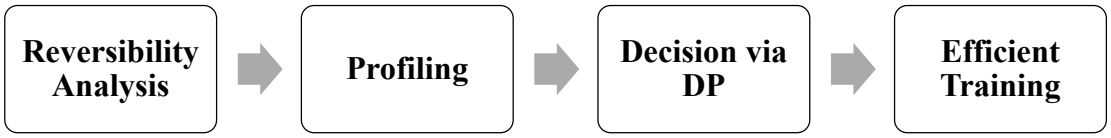


Figure 3.8: Four stages in our framework

Therefore, we can minimize the training time by minimizing the total wall-clock time of all these reversible layers.

3.2.3.3 Algorithm and Framework

Problem 3.11 can be rewritten as Problem 3.12.

$$\begin{aligned} \max_{x} \quad & t_e^T x \\ \text{s.t.} \quad & m^T x \leq M - m_o \\ & x_i \in \{0, 1\}, i = 1, ..., n \end{aligned} \tag{3.12}$$

Problem 3.12 can be interpreted as follows. We take the `baseline-M` ($x = \mathbf{0}$) as the reference. The object function $t_e^T x$ is the execution time reduction when we apply the decision x. The remaining memory capacity for these reversible layers is $M - m_o$.

Problem 3.12 is a standard **0/1 knapsack problem** in essence [130]. Note that the memory-related variables and parameters m, M, m_o are all positive integers since all of them are in the unit of bytes. Therefore, it can be solved by dynamic programming, as shown in Algorithm 3.

Based on the algorithm, we propose a framework to automate the decision process. Figure 3.8 shows the four stages of our framework. Initially, we verify the reversibility of each operator. The correctness of the original and inverse functions will be verified. In the second stage, we will obtain parameters t_e and m from realistic measurements. Our framework is

Algorithm 3 Dynamic programming algorithm for 0/1 knapsack problem

Input: $t_e, m, M - m_o, n$ ▷ Indices of vectors t_e and m start from 1.

Define saved$[n, M-m_o]$ and initialize all entries as -1, which means the entry is undefined.

▷ The entry saved$[i, j]$ records the maximum saved time under the condition that we consider first i items with total memory limit of j.

foo(i, j) ▷ This recursive function calculates saved$[i, j]$.

  if $i == 0$ **or** $j \leq 0$ **then**

    return ▷ No time saved under this condition

  end if

  if saved$[i-1, j] == -1$ **then**

    saved$[i-1, j] =$ **foo**$(i-1, j)$

  end if

  if $m[i] > j$ **then**

    saved$[i, j] =$ saved$[i-1, j]$

  else

    if saved$[i-1, j-m[i]] == -1$ **then**

      saved$[i-1, j-m[i]] =$ **foo**$(i-1, j-m[i])$

    end if

    saved$[i, j] = \max\{$saved$[i-1, j],$ saved$[i-1, j-m[i]] + t_e[i]\}$

  end if

  return saved$[i, j]$

end foo

saved$[n, M-m_o] =$ **foo**$(n, M-m_o)$

Initialize decision variables $x =$ **0** ▷ Do backtracking to find the optimal solution.

$j = M - m_o$

for $i = n, n-1, ..., 1$ **do**

  if saved$[i, j] \neq$ saved$[i-1, j]$ **then**

    $x[i] = 1$

    $j = j - m[i]$

  end if

end for

return saved$[n, M-m_o], x$ ▷ Return optimal values and solutions.

hardware-aware since we use realistic profiling data from specific machines. Then we use Algorithm 3 to get the optimal solution. Finally, we can train the network with maximum throughput. The dynamic programming algorithm will only be executed once to obtain the optimal schedule. After that, this schedule can be used in all the training iterations. Thus, the added complexity is negligible compared with the training process.

3.2.3.4 Various Mini-batch Size

The above discussions are based on the assumption that the mini-batch size is fixed. When we have many choices on the mini-batch size (denoted b), the optimization problem will be more complicated.

We assume that for each layer, its execution time is linear to the batch size, whether it is reversible or not. Namely, the execution time satisfies that $t(b) = t^{(0)} + bt^{(1)}$. The total execution time of all the non-reversible layers is $t_n^{(0)} + bt_n^{(1)}$. The total execution time of all the reversible layers is

$$\mathbf{1}^T(t_{f1} + t_{b1}) - t_e^T x = \mathbf{1}^T(t_{f1}^{(0)} + t_{b1}^{(0)}) + b\mathbf{1}^T(t_{f1}^{(1)} + t_{b1}^{(1)}) - t_e^{(0)^T} x - bt_e^{(1)^T} x$$

The execution time of the optimizer, scheduler, and control are independent of mini-batch size, denoted by t_o. The execution time per sample is

$$t_n^{(1)} + \mathbf{1}^T(t_{f1}^{(1)} + t_{b1}^{(1)}) - t_e^{(1)^T} x + \frac{t_o + t_n^{(0)} + \mathbf{1}^T(t_{f1}^{(0)} + t_{b1}^{(0)}) - t_e^{(0)^T} x}{b}$$

The memory footprint is also linear to the mini-batch size. The size of network parameters is independent of the mini-batch size. The size of the feature maps of the non-reversible layers is proportional to the mini-batch size. Thus, the memory constraint can be rewritten as $bm^T x + m_o^{(0)} + bm_o^{(1)} \leq M$.

The optimization problem is now

$$\begin{aligned} \min_{x,b} \quad & t_n^{(1)} + \mathbf{1}^T(t_{f1}^{(1)} + t_{b1}^{(1)}) - {t_e^{(1)}}^T x + \frac{t_o + t_n^{(0)} + \mathbf{1}^T(t_{f1}^{(0)} + t_{b1}^{(0)}) - {t_e^{(0)}}^T x}{b} \\ \text{s.t.} \quad & bm^T x + m_o^{(0)} + bm_o^{(1)} \leq M \\ & x_i \in \{0,1\}, i = 1, ..., n \\ & b \in [b_l, b_u], b \in \mathbb{Z} \end{aligned} \tag{3.13}$$

where b_l, b_u are lower and upper bounds of the mini-batch size.

Rewrite the problem as Problem 3.14.

$$\begin{aligned} \max_{x,b} \quad & f(x,b) = {t_e^{(1)}}^T x - \frac{C - {t_e^{(0)}}^T x}{b} \\ \text{s.t.} \quad & bm^T x + m_o^{(0)} + bm_o^{(1)} \leq M \\ & x_i \in \{0,1\}, i = 1, ..., n \\ & b \in [b_l, b_u], b \in \mathbb{Z} \end{aligned} \tag{3.14}$$

where $C = t_o + t_n^{(0)} + \mathbf{1}^T(t_{f1}^{(0)} + t_{b1}^{(0)})$ is a constant.

Problem 3.14 is a non-linear integer programming optimization problem, which is hard to get the optimal solution. A simple method is to sweep the mini-batch size in the range of $[b_l, b_u]$ with our framework. Empirically the Problem 3.12 is fast to solve using Algorithm 3. Thus, it is affordable to apply the simple method of sweeping the mini-batch size. We further discuss various mini-batch size in in Section 3.2.4.6. We leave Problem 3.14 as an open problem for future research.

3.2.4 Experiments

In this section, we provide the experimental settings initially. Then we discuss the details of profiling. We analyze three reversible neural architectures: RevNet-104, ResNeXt-

101 with inplace ABN, and Reformer. We further discuss the results in terms of various mini-batch sizes.

3.2.4.1 Settings

We adapt source codes from MemCNN[5] [108], Inplace ABN[6] [23], and Reformer[7] [95]. We follow their original settings and hyperparameters except that we can decide what modes each reversible layer will use.

Unless otherwise stated, we use PyTorch [142] 1.4.0. The training process runs on a Linux server with Intel Core i9-7900X CPU and 1 NVIDIA TITAN Xp GPU, whose memory capacity is 12,196 MiB. All the tensor operations are on the GPU. We report the mean of 100 training iterations.

3.2.4.2 Profiling

To ensure hardware-awareness, our framework needs to do profiling on the execution time and memory allocation to obtain t_e, m, m_o based on realistic measurement. It is easy to collect memory-related terms m, m_o since the memory footprint is stable throughout a whole training process.

For the execution time t_e, the most accurate way to obtain it is running the model in two modes respectively and collect all the four corresponding vectors (t_{f1}, t_{b1}, t_{f2}, t_{b2}). We can also directly compare these two modes and conclude their difference. For the feature maps in the `C-Mode`, it takes extra time for the memory writes in the forward computation,

[5]https://github.com/silvandeleemput/memcnn
[6]https://github.com/mapillary/inplace_abn
[7]https://github.com/lucidrains/reformer-pytorch

and memory read in the backward pass. In the `M-Mode`, there is overhead in reading y from memory and the inverse computation.

It is complicated to analyze the memory behaviour, and the analysis is beyond discussions of this dissertation. Fortunately, we observe that $t_{f1} \approx t_{f2} \approx t_{b1} - t_{b2}$. For instance, the average execution time of the RevNet-104 [56] with mini-batch size of 64 on ImageNet is $t_{f1} = 10.425$ms, $t_{f2} = 10.404$ms, $t_{b1} = 29.276$ms, and $t_{b2} = 18.865$ms. This observation is prevalent in current machine learning frameworks, since memory accesses are hidden by computations [1, 142]. Thus, we can only take computation into account when analyzing the difference in execution time. In short, $t_e = (t_{f1} + t_{b1}) - (t_{f2} + t_{b2}) \approx t_{f1} \approx t_{f2}$ The assumption is verified for all the following experiments. We use $t_e = t_{f1}$ in the optimization problem directly.

3.2.4.3 RevNet

We apply our framework on RevNet-104 [56] for image classification on ImageNet. By sweeping the mini-batch sizes, we can obtain various memory budgets and computation overhead. Figure 3.9 illustrates our decision for different mini-batch sizes. When the mini-batch size is smaller than 65, the GPU memory capacity is large enough to contain all the intermediate activations. Thus, the optimal decision is saving all of them to achieve maximum training throughput. Starting from a mini-batch size of 65, we have to use the `M-Mode` in partial reversible layers due to the limited memory budget. Our dynamic programming solver will obtain the optimal decision for each setting. If the mini-batch size is larger than 117, we will encounter the issue of out of memory even if we use `baseline-M`, the most memory-efficient decision. As shown in Figure 3.9, the optimal decision is non-trivial across

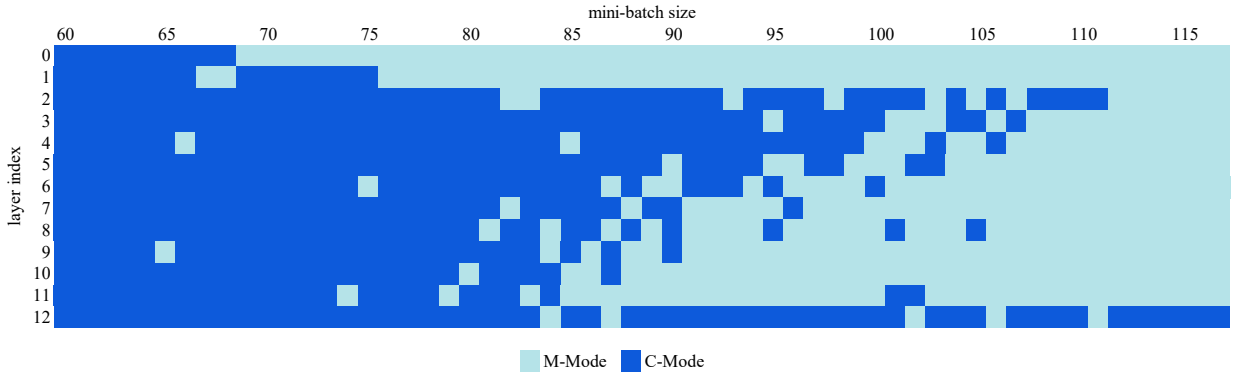
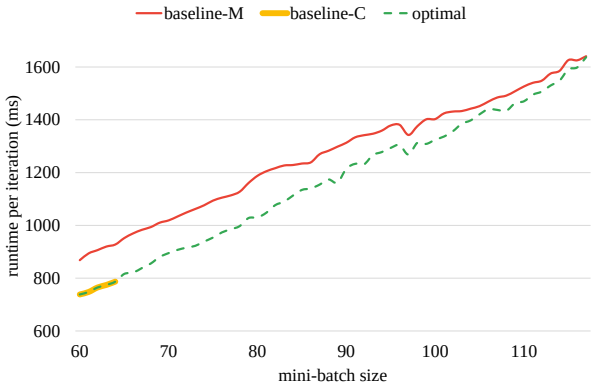


Figure 3.9: The heat map of the optimal solutions throughout different mini-batch sizes on RevNet-104 with 13 reversible layers. The horizontal and vertical axes represent the mini-batch size and the layer index, respectively.

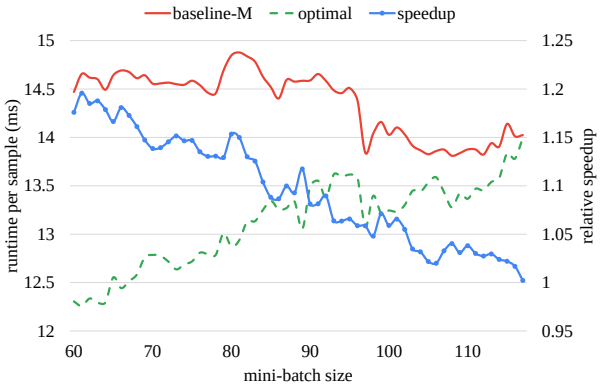
different mini-batch sizes.

Figure 3.10a shows the training time per iteration of `baseline-M`, `baseline-C` and our optimal solution. The solid red line and the green dashed line represents the `baseline-M` and optimal settings provided by our framework. The `baseline-C` is highlighted in the lower left corner, since it is limited by the device's memory capacity and cannot contain a large batch size. Our optimal solution overlaps with the `baseline-C` when `baseline-C` is feasible, i.e., mini-batch size smaller than 65. When `baseline-C` is not available, our framework approach the `baseline-M` gradually. The reason is that as the mini-batch size grows, the harsh memory constraint pushes us forward to the extreme of memory efficiency. The gap between the two curves (`baseline-M` and optimal) demonstrates the absolute time saved by applying our method.

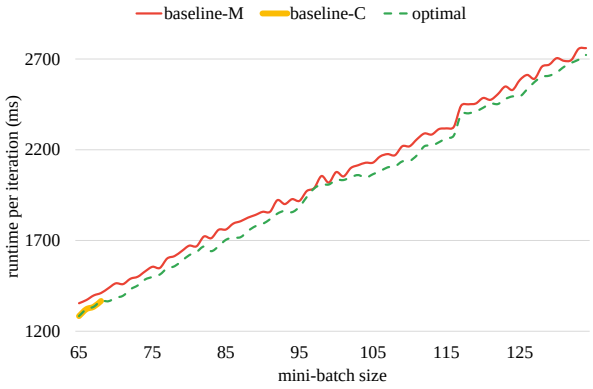
Figure 3.10b compares the training time per sample. We can use this metric to compare the training throughput (which is the multiplicative inverse of the training time per



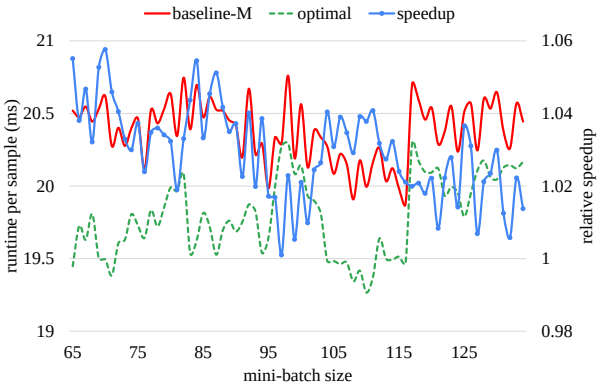
a Training time per iteration of RevNet-104



b Training time per sample and relative speedup of RevNet-104



c Training time per iteration of ResNeXt-101 with Inplace ABN



d Training time per sample and relative speedup of ResNeXt-101 with Inplace ABN

Figure 3.10: Training time and speedup comparison of RevNet-104 and ResNeXt-101 with Inplace ABN on ImageNet. Training time per iteration is the time of one complete iteration (forward, backward, and optimizer updating). Training time per sample is the multiplicative inverse of training throughput. The curves of `baseline-C` are truncated due to device memory limitation.

sample) for different mini-batch sizes. Before applying our framework, the training speed increases as the mini-batch size grows for two reasons. First, we leverage the parallelism across batches. Second, the execution time of the optimizer, scheduler, and control is independent of the mini-batch size. This part of execution is amortized by the large mini-batch size. After using our framework, the trend is different. The training throughput decreases as the mini-batch size grows, because the computation overhead of inverse functions is much larger than the benefit from large mini-batch size. We also show the relative speedup of our optimal execution time compared with `baseline-M`. We can achieve up to 1.15× speedup for this benchmark.

3.2.4.4 Inplace ABN

We follow the settings in the paper of Inplace ABN [23] and use our framework to train ResNeXt-101 [199] for image classification on ImageNet. Figures 3.10c and 3.10d compares the training time per iteration across different mini-batch sizes. The results are similar to those of RevNet-104 except the relative speedup.

The computation overhead of the Inplace ABN is relatively low compared with RevNet-104 in the previous section. The execution time of `baseline-C` is only $0.8 - 2\%$ smaller than that of `baseline-M`. Therefore, the relative speedup using our method is not as significant as the experiments on RevNet-104. The reason is that the maximum training throughput of our framework is bounded by `baseline-C`. However, the advantage of our method is that we can find the optimal point across two baselines.

mini-batch size	baseline-C TPI	baseline-M TPI	optimal TPI	baseline-C TPS	baseline-M TPS	optimal TPS	speedup
1	0.951	1.321	0.949	0.951	1.321	0.949	1.392
2	1.738	2.533	1.738	0.869	1.266	0.869	1.457
3	OOM	3.603	2.752	OOM	1.201	0.917	1.310
4	OOM	4.792	4.175	OOM	1.198	1.044	1.148
5	OOM	6.020	5.236	OOM	1.204	1.047	1.150
6	OOM	7.210	6.692	OOM	1.202	1.115	1.077
7	OOM	8.420	7.670	OOM	1.203	1.096	1.098
8	OOM	9.490	9.044	OOM	1.186	1.130	1.049
9	OOM	10.603	10.123	OOM	1.178	1.125	1.047
10	OOM	11.873	11.295	OOM	1.187	1.129	1.051

Table 3.3: Results of Reformer on enwik8 task. TPI and TPS are abbreviations for training time per iteration and training time per sample. OOM stands for out of memory. All the execution time is in the unit of seconds.

3.2.4.5 Reformer

We also do experiments on the enwik8 task with Reformer. Specifically, there are 8 heads in our 12-layer model. The maximum sequence length is 4,096, and the number of tokens is 256. For each iteration, we call the optimizer to update the trainable parameters after accumulating gradients for 4 steps. Table 3.3 shows the training time in different modes.

Due to the large memory footprint, the `baseline-C` can only run with a mini-batch size of 2. The reversibility enables us to train the model with a mini-batch size up to 10. Our framework provides a smooth transition from `baseline-C` to `baseline-M`. We achieve 1.3× relative speedup when the mini-batch size is 3.

3.2.4.6 Various Mini-Batch Sizes

For this section, we discuss the optimal mini-batch size from the perspective of training throughput. In the above experiments, the lowest execution time per sample (TPS) is approximately obtained at the largest mini-batch size when `baseline-C` is feasible. For example, the Reformer get the lowest TPS 0.869s at the mini-batch size of 2. The reason is that the computation overhead of inverse functions is much larger than the benefit from large mini-batch size. In other words, we cannot accelerate the training process via reversible neural architectures. From the perspective of Problem 3.14, the TPS $f(x, b) = {t_e^{(1)}}^T x - \frac{C - {t_e^{(0)}}^T x}{b}$ is dominated by the first term ${t_e^{(1)}}^T x$.

3.2.5 Summary

We present the framework to train reversible neural architectures in the most efficient schedule. We formulate the decision problem for reversible operators. The training time is the objective function with memory usage as a constraint. By solving this problem, we can maximize the training speed for any reversible neural architectures. Our framework automates this decision process, empowering researchers to develop and train reversible networks more efficiently.

For future directions, we may integrate gradient checkpoints and reversible neural architectures to enlarge the search space, since gradient checkpoints allow non-reversible layers to follow M-Mode by doing recomputation. The optimal mini-batch size in terms of training throughput is another critical issue.

3.3 Optimizer Fusion: Efficient Training with Better Locality and Parallelism

Iterative methods, such as stochastic gradient descent (SGD) and its variants, are the mainstream optimization algorithms for training machine learning models. In the optimization process of those algorithms, learnable parameters are updated step by step until the stopping criterion is met. Many commonly used iterative optimization methods are implemented in popular machine learning frameworks, e.g. PyTorch [142], TensorFlow [1], MXNet [31], and Chainer [178].

One critical component of these machine learning frameworks is automatic differentiation, which computes the gradients for all operations on tensors. The smooth integration between the optimization kernel and automatic differentiation makes the training more accessible and boosts the popularization of these frameworks in the machine learning community.

Eager execution is widely adopted in the machine learning frameworks for its flexibility. It usually decouples the forward propagation, gradient computation, and parameter updating into three separate stages. In each iteration, forward computation is first performed. Gradients respective to the loss function are then calculated for all learnable parameters. Finally, learnable parameters are updated by a specified optimizer. Although this implementation has an intuitive and transparent procedure, the learnable parameters and their gradients are read and written several times throughout one training iteration, such that these data are not efficiently reused. Moreover, gradient updating is enforced to be after the gradient computation based on the implicit control dependency, leading to lower parallelism in the program execution. In short, there is potential for higher training efficiency with better locality and parallelism in the eager execution.

3.3.1 Overview

We propose two methods `forward-fusion` and `backward-fusion`, which reorder the forward computation, gradient calculation, and parameter updating to accelerate the training process in eager execution. Our proposed methods fuse the optimizer with forward or backward computation to better leverage the locality and parallelism. The `backward-fusion` method, motivated by the static computational graph compilation, where the optimizer is fused with gradient computation, updates the parameters as early as possible. [8] The `forward-fusion` method fuses the parameter updates with the next forward computation such that learnable parameters are updated as late as possible. Like back-propagation through time (BPTT) [190], the `forward-fusion` method expands the training process through iterations and uncovers a novel perspective of acceleration.

We summarize the advantages of our methods as follows.

- Efficient. Our framework can increase the training speed by 15% on average.
- General. Our methods are orthogonal to other optimization methods and do not affect the training results. Thus, they can be applied in the training process of various machine learning tasks with different optimizers. We keep all the features of the eager execution.
- Simple. It is easy to replace the old training routines with our new methods. Users can accelerate their imperative training with little engineering effort.

[8] For example, the optimizer in TensorFlow supports different gating gradients configurations. https://www.tensorflow.org/api_docs/python/tf/compat/v1/train/Optimizer#gating_gradients

3.3.2 Background

In this section, we present the general structure of iterative optimization methods. Moreover, we discuss how locality and parallelism are considered to accelerate the training process. Finally, we review two machine learning programming paradigms: symbolic and imperative programs.

3.3.2.1 Iterative Optimization Methods

An iterative optimization algorithm starts from an initial guess and derives a sequence of improving approximate solutions. Algorithm 4 is the general structure of iterative optimization methods for unconstrained problems. The step vector $\Delta\theta$ is computed from the optimizer policy π. The policy π is the only difference across different optimization algorithms. The commonly used gradient-based methods use policies that depend on the first-order derivative information.

For instance, the policies demonstrated in Equations equation 3.15, equation 3.16, and equation 3.17 are used in the gradient descent, gradient descent with momentum, Newton's method, respectively,

$$\pi_1 = -\eta \nabla f(\theta^{(t-1)}) \tag{3.15}$$

$$\pi_2 = -\eta \sum_{\tau=0}^{t-1} \alpha^{t-\tau-1} \nabla f(\theta^{(\tau)}) \tag{3.16}$$

$$\pi_3 = -\eta \nabla^2 f(\theta^{(t-1)})^{-1} \nabla f(\theta^{(t-1)}) \tag{3.17}$$

where η represents the step size, α denotes the momentum decay factor.

Algorithm 4 Optimization algorithms in general form

Input: objective function f
Initialize the starting point $\theta^{(0)}$
for $t = 1, 2, ...$ **do**
  if stopping criterion is met **then**
    return $\theta^{(t-1)}$
  end if
  $\Delta\theta = \pi(f, \theta^{(0)}, \theta^{(1)}, ..., \theta^{(t-1)})$
  $\theta^{(t)} = \theta^{(t-1)} + \Delta\theta$
end for

3.3.2.2 Locality and Parallelism

Fruitful hardware-aware techniques in machine learning have been proposed to accelerate the training process, especially on graphics processing units (GPUs). Generally speaking, common approaches include accelerating the kernel computations [112], mixed precision training [133] , fusing kernels and operations [195], and exploring efficient network architecture [57]. Data locality and computation parallelism are two critical aspects of performance optimization.

The work of fused-layer CNN accelerators [4] proposes a new architecture for inference by fusing the convolution layers. It decomposes the inputs to the convolution layers into tiles and propagates one tile through multiple layers. With reduced memory access and better cache utilization, the inference speed is improved. Lym et al. [123] design a new scheme to eliminate most memory accesses in neural network training by reordering the computation within a mini-batch for better data locality. Apex for PyTorch [9] uses fused optimizers, which launch one kernel for the element-wise operations.

[9] https://github.com/NVIDIA/apex

From a perspective different from the aforementioned previous works, we explore the parallelism and locality across the gradient computation and the optimizer. We also uncover the potential of acceleration across iterations.

3.3.2.3 Static and Dynamic Computational Graphs

The training process can be viewed as a computational graph. Figure 3.11(a) demonstrates the corresponding computation dependency for a three-layer neural network, where nodes represent tensor computations, and directed edges stand for data dependencies. Any topological order of this graph is a valid computation order.

Generally, there are two paradigms for tensor computations in machine learning frameworks. The first category of framework compiles a machine learning model as a *static (symbolic) computational graph* and executes the graph with all the neural network information. For example, TensorFlow 1.X follows this routine by default [1], which requires a pre-compiled graph before execution. The other category of framework works in the *eager (imperative) mode*, which immediately executes the newly-encountered computation node and incrementally builds a dynamic computational graph with that node. At each step, a computation node will be appended to the current computational graph. PyTorch [142] and TensorFlow 2.X [1] both run in eager mode by default, enabling users to develop machine learning models more easily and quickly. The eager mode also enables the efficient development of non-stationary neural architectures.

These two categories of frameworks are both widely used in the machine learning community. For example, in the MLPerf training benchmark [132], both static and eager modes are used and achieve the state of the art performance.

Engineers and practitioners in machine learning frameworks have proposed and implemented many *graph optimizers*, such as TensorFlow's Grappler [103] and TASO [82]. The `backward-fusion` method has been considered in some of them. However, to our best knowledge, existing graph optimizers need the information of the whole computation graph, which means the optimizer fusion can only be used for (1) the static computation or (2) the mixture of static and dynamic execution. In most machine learning frameworks, the purely eager mode still separates forward computation, backward pass, and parameter updating into three stages, with the topological order shown in Figure 3.11(b). The framework will first execute the forward and backward computation to obtain gradients. Then learnable parameters will be updated following the optimizer. This imperative nature allows users to monitor the training process at the cost of low efficiency. For instance, the TensorFlow eager execution follows this routine. [10]

We enable the `backward-fusion` method in purely eager mode. The `forward-fusion` method also uncovers a novel perspective, where graphs can be optimized across iterations.

3.3.3 Methods

We first analyze the baseline in terms of locality and parallelism. Then, to improve the training efficiency, we propose two methods, denoted `forward-fusion` and `backward-fusion`. Both methods reorder the original computation graph to better leverage data locality and computation parallelism.

[10] https://www.tensorflow.org/api_docs/python/tf/compat/v1/train/Optimizer#minimize

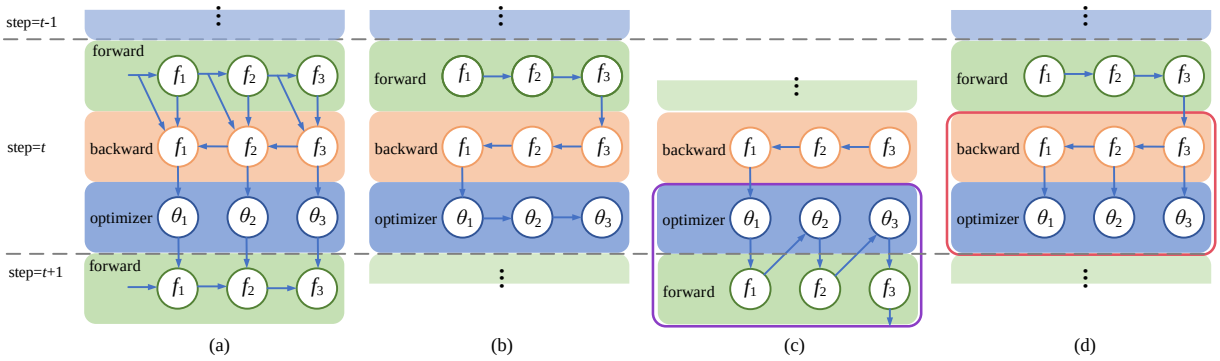


Figure 3.11: (a) Data dependency graph. (b) Baseline method. (c) `Forward-fusion`. (d) `Backward-fusion`. θ_i represents the trainable parameters in the layer f_i.

3.3.3.1 Baseline

Figure 3.12 illustrates the memory transactions and the locality that can be leveraged in the training process. Trainable parameters are read during forward and backward computations and updated by the optimizer. Gradients are accumulated during the backward pass. Finally, they are read and reset by the optimizer. History represents the parameter history, such as momentum. The optimizer records and updates the parameter history.

In the baseline method, all these memory reads and writes are separated by forward, backward, and optimizer stages. The memory capacity is usually not large enough to hold all the data through the training iteration. The data locality between the optimizer and its adjacent forward or backward computations is lost. If we access the same set of data repetitively before the data is flushed, we can shorten the time of memory access and thus accelerate the training process.

Also, the baseline method does not take advantage of the parallelism between backward computations and parameter updating. While updating a group of parameters, we can

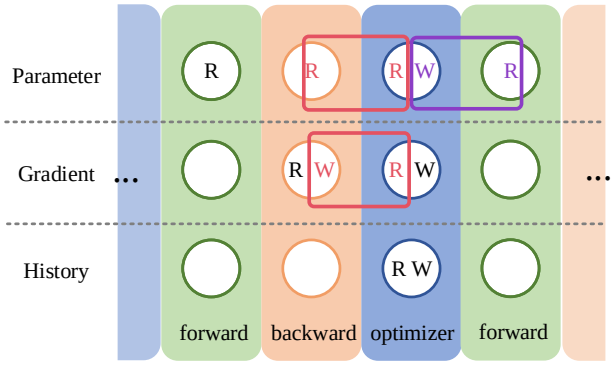


Figure 3.12: Memory transactions and data locality in the training process. R and W represent memory read and write, respectively. *History* means parameter history needed in the optimizer, e.g., momentum. Red and purple frames represent locality improvement in `backward-fusion` and `forward-fusion`, respectively.

continue the back-propagation to compute gradients for other independent parameters at the same time, which offers another opportunity for better parallelism in the training process.

3.3.3.2 Forward-Fusion

One approach is to fuse the optimizer with the forward computation in the next iteration. The next forward pass can occur in either a training or an evaluation process. Each trainable parameter will be updated as late as possible. This *lazy* update strategy is named `forward-fusion`. The proposed method can be applied in all the iterative methods, including the optimizer that needs global information.

Figure 3.11(c) and Algorithm 5 shows the pipeline of this method. It is possible that a layer is used many times, i.e., $f_i = f_j, i \neq j$ in Algorithm 5. We use a flag `updated` to ensure that the parameter is updated only once no matter how many times the corresponding layer is used. This method can also be applied when we need to manipulate the gradients

based on global information [154]. For instance, this method is suitable when we would like to clip the gradients by their global norm.

Algorithm 5 Forward-fusion

```
Input: topologically sorted operator array {f_i}_{i=1}^n
for i = 1, 2, ..., n do
    if f_i.updated is False then
        for each trainable parameters θ in f_i do
            execute optimizer of θ
        end for
        f_i.updated ← True
    end if
    execute f_i
end for
for i = n, n − 1, ..., 1 do
    execute backward pass of f_i
    accumulate gradients for all trainable parameters
    f_i.updated ← False
end for
```

The memory write operation during parameter updating can be merged with the next read, such that in the next forward computation, the cached parameter can be quickly accessed with low latency. The purple frame in Figure 3.12 illustrates this improvement.

Unlike current machine learning frameworks, which ignore the optimization potential across different iterations, we find the opportunity between adjacent training steps. Similar to back-propagation through time [190], the training process of a feed-forward neural network training can also be expanded through iterations. This perspective is a new direction for accelerating the machine learning models embracing both static and dynamic computational graphs.

3.3.3.3 Backward-Fusion

Another approach is to fuse the optimization computation with gradient computation in the backward pass. Figure 3.11(d) and Algorithm 6 demonstrates the computational flow. After calculating the gradient of θ_i, we apply the gradient in the optimizer directly. At the same time, we resume the backward computation for node f_{i-1}. For each parameter θ_i, we record the number of its usage in the forward pass as θ_i.count. Correspondingly, we will update it until its gradient is accumulated for all its usage in the forward pass.

Algorithm 6 Backward-fusion

```
Input: topologically sorted operator array {f_i}_{i=1}^n
for i = 1, 2, ..., n do
    execute f_i
    for each trainable parameters θ in f_i do
        θ.count ← θ.count +1
    end for
end for
for i = n, n − 1, ..., 1 do
    execute backward pass of f_i
    for each trainable parameters θ in f_i do
        accumulate gradient of θ
        θ.count ← θ.count −1
        if θ.count is 0 then
            execute optimizer of θ
        end if
    end for
end for
```

Applying gradients directly on the parameters may induce race conditions. For instance, for a multiplication operator $f(\theta, x) = \theta x$, $\partial f/\partial \theta = x, \partial f/\partial x = \theta$. $\partial f/\partial x$ depends on the original parameter $\theta^{(t)}$, instead of the updated parameter $\theta^{(t+1)}$. Therefore, we must carefully tackle this dependency. Specifically, for a trainable parameter θ, we will update it

Method	Locality	Parallelism	Global Info.
baseline	×	×	✓
forward-fusion	✓	×	✓
backward-fusion	✓	✓	×

Table 3.4: Comparison among three methods: locality, parallelism, and global information.

in-place to obtain $\theta^{(t+1)}$ when the following two conditions are both satisfied: (1) its gradient $\partial L/\partial\theta$ is calculated, and (2) there is no other dependency on the old value $\theta^{(t)}$.

This method applies gradients to parameters as early as possible, so that the memory access can be merged to increase the locality, as shown in the red frame in Figure 3.12. Specifically, two consecutive parameter reads in the backward pass and optimizer can be merged, such that the second read during the optimization step can be accelerated as it can be cached in the local storage. Gradient accumulation in the backward computation can be merged with the memory read in the optimizer. Thus, the updated gradients can be efficiently accessed in the local storage to shorten the memory access time.

Moreover, this method improves training efficiency as it parallelizes the parameter updating and gradient back-propagation. This method applies to most optimizers that do not require global information of trainable parameters, as this fusion strategy assumes the update of θ_i is decoupled with other parameters $\theta_j (i \neq j)$.

In the view of the computational graph, the depths of the directed graphs shown in Figures 3.11(b), (d) are $3n$ and $2n + 1$, respectively, where n is the number of layers in the neural network. Thus, the `backward-fusion` method also provides extra parallelism.

Table 3.4 summarizes the characteristics of our proposed methods.

3.3.4 Experiments

We evaluate the effectiveness and efficiency of our proposed methods with various mini-batch sizes, optimizers, machine learning models and benchmarks, machines (GPUs) and frameworks.

3.3.4.1 Experimental Settings

Unless stated otherwise, we conduct experiments on the eager execution in PyTorch 1.6.0. We implement the proposed methods in the PyTorch front-end using hooks. A toy example is provided in the supplementary file. The training process runs on a Linux server with Intel Core i9-7900X CPU and a NVIDIA TITAN Xp GPU based on Pascal architecture. We use Adam [92] with weight decay to do the training on image classification using the ImageNet dataset [43]. All the tensor computations occur on 1 GPU with single-precision floating-point (float32) datatype. We report the mean of 100 training iterations.

3.3.4.2 Runtime Breakdown

Figure 3.13 shows the execution time breakdown of one training iteration of MobileNetV2 with a mini-batch size of 32. After we fuse the optimizer with backward computation, the execution time of backward increases by 3.32 ms, much smaller than the original optimizer execution time (16.70 ms). In this example, our `forward-fusion` and `backward-fusion` improve the training throughput by 12% and 16%, respectively, which demonstrates the effectiveness of our methods.

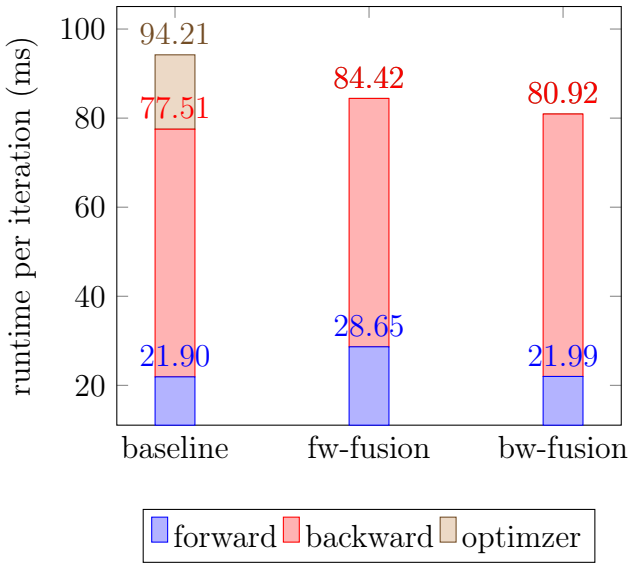


Figure 3.13: Training time breakdown of MobileNetV2 with mini-batch size 32. Our `forward-fusion` and `backward-fusion` methods improve the training throughput by 12% and 16%, respectively.

3.3.4.3 Various Mini-batch Sizes

Compared with the baseline method, our two methods have the overhead of control as shown in Algorithms 5 and 6. When a small mini-batch size is used, the overhead of the control flow will exceed the benefits of better locality and parallelism, which makes our framework slower than the baseline. As the mini-batch size increases, the overhead becomes negligible compared with the computation time. The effectiveness of our proposed methods requires this overhead to be amortized by an appropriate mini-batch size.

When the mini-batch size is large enough such that we reach the performance roofline of the GPUs, the computation time of forward and backward pass is approximately linear to the mini-batch size. On the other hand, the optimizer execution time is independent of the mini-batch size. Therefore, the absolute training time saved by our methods is independent

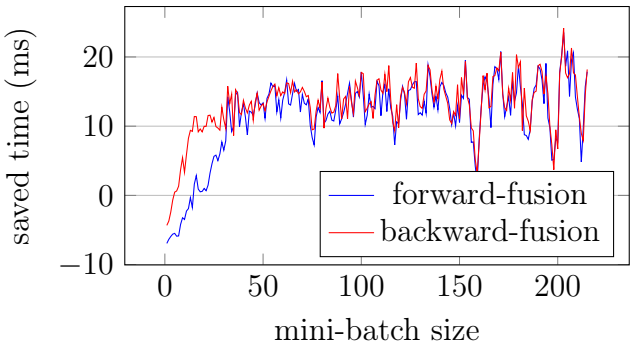


Figure 3.14: The absolute execution time saved by our methods on MobileNetV2 with different mini-batch sizes.

of the mini-batch size, as shown in Figure 3.14. The relative speedup will decrease as the mini-batch size grows, as demonstrated in Figure 3.15.

The discussion above can also be formulated in the following equation. The theoretical speedup of the training process is

$$s = \frac{bt_{grad} + t_{opt}}{bt_{grad} + t_{opt} - t_{saved}}$$

where b represents the mini-batch size, t_{grad} is the time of forward and backward computation per mini-batch size, t_{opt} is the execution time of optimizer, and t_{saved} stands for the absolute saved time on the optimizer with our methods.

Both `forward-fusion` and `backward-fusion` methods leverage the locality of the trainable parameters. However, only the `backward-fusion` method takes advantage of the parallelism between the gradient computation and optimizer. When the mini-batch size is small, the GPU is not fully utilized. Thus, the parallelism exploited by `backward-fusion` will accelerate the training significantly compared with `forward-fusion`. As mini-batch sizes grow, the gradient computation dominates the GPU utilization. Therefore, the execution

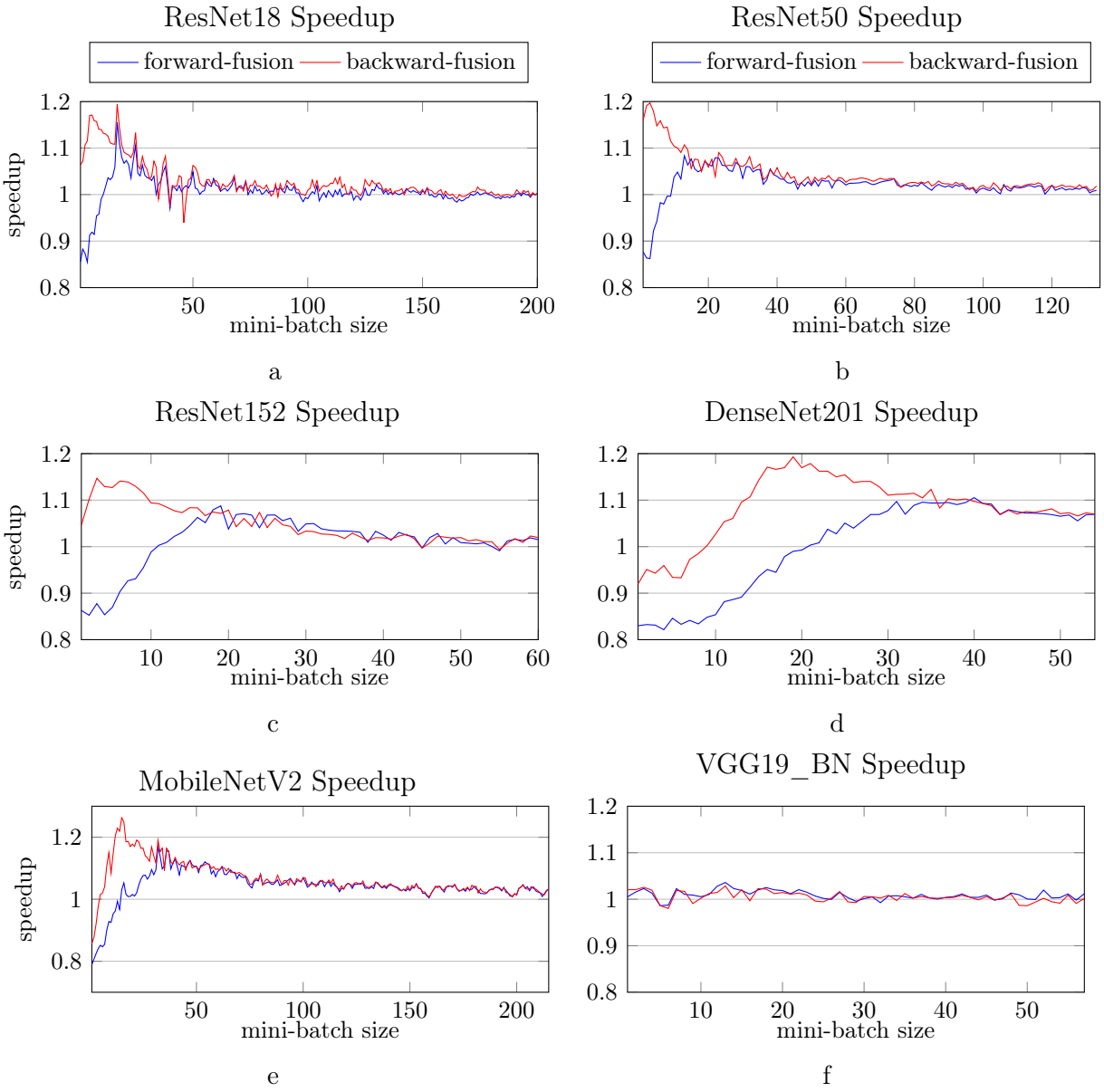


Figure 3.15: Training speedup with various mini-batch sizes on different benchmarks.

time of these two methods converges at large mini-batch sizes, as illustrated in Figure 3.15.

3.3.4.4 Various Models and Optimizers

We sweep the mini-batch size for different models [63, 71, 76, 161, 167] as shown in Figure 3.15. Figure 3.16 demonstrates the relationship between the parameter size and speedup across different models. The smaller the average number of parameters per layer, the more locality we can leverage so that our methods can achieve higher training speed. This explains why the VGG19_BN is hardly accelerated while the MobileNetV2 has the most significant improvement. Currently, the models targeting at edge devices usually contain fewer parameters, whose training will benefit more from our methods.

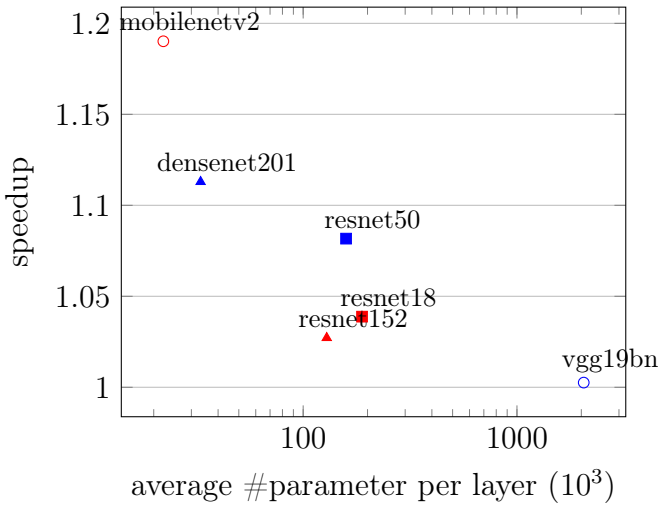


Figure 3.16: The speedup trend among different models with a mini-batch size of 32. On average, fewer parameters per layer leads to higher speedup.

Various optimizers used in machine learning frameworks can benefit from our proposed methods. Figure 3.17 shows an increasing trend between speedup and the runtime

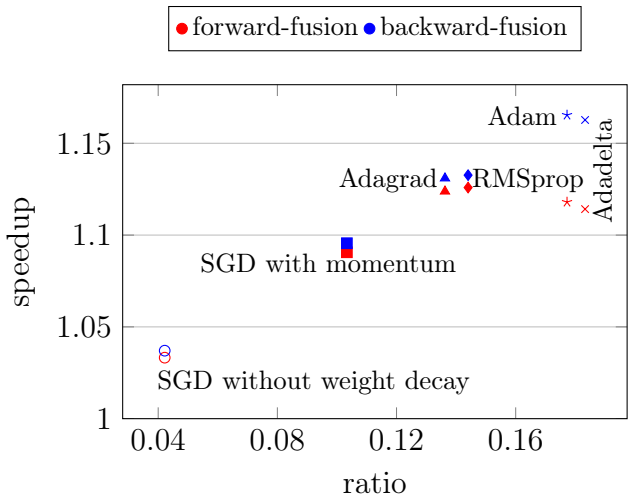


Figure 3.17: Comparison among various optimizers on MobileNetV2 with a mini-batch size of 32. Weight decay is applied in all these methods unless specified. The horizontal axis represents the ratio of the optimizer time to a whole iteration time.

ratio of different optimizers [48, 92, 207]. The horizontal axis is the ratio of the optimizer runtime to a whole iteration runtime. The more runtime-costly the optimizer, the higher speedup we can achieve.

3.3.4.5 Various Machines and Benchmarks

CPU	GPU	baseline runtime (ms)	forward-fusion runtime (ms)	backward-fusion runtime (ms)	forward-fusion speedup	backward-fusion speedup
Core i9-7900X	TITAN Xp	98.77	84.52	82.99	1.17	1.19
Core i7-3770	GTX 1080	163.60	145.80	129.71	1.12	1.26
Core i7-8750H	GTX 1070 maxQ	174.43	157.27	158.89	1.11	1.10

Table 3.5: Training results on MobileNetV2 with a mini-batch size of 32 across various machines.

Our methods are practical and efficient on various machine configurations, as shown in Table 3.5. The speedup depends on the cache size, the floating point operations per

second (FLOPS), memory bandwidth, etc. Although the relationship is very complicated and beyond our discussion, our methods stay effective on various GPUs.

Our methods can be used in all the iterative optimization problems. Thus it can be used in almost all machine learning problems. For example, we train the Transformer (base) [183] on the WMT English-German dataset. With a min-batch size of 256, we can achieve the speedup of 1.030 and 1.019 respectively using our `forward-fusion` and `backward-fusion` methods.

3.3.5 Summary

Conventional eager execution in machine learning frameworks separates the updating of trainable parameters from forward and backward computations. We propose two methods `forward-fusion` and `backward-fusion` to better leverage the locality and parallelism during training. We reorder the forward computation, gradient calculation, and parameter updating so that our proposed methods improve the efficiency of iterative optimizers. Experimental results demonstrate the effectiveness and efficiency of our methods across various configurations. Our `forward-fusion` method opens a new perspective of performance optimization for machine learning frameworks.

For future directions, we plan to extend our methods to distributed training [111], where there is parallelism across different machines. The gradient computation, reduction, and parameter updating can be merged to save time.

3.4 Summary of the Chapter

Three efficient compilation methods are presented: removing redundancy in Pre-LN Transformers, solving the scheduling problem for reversible neural networks, and employing optimizer fusion to consider memory hierarchy locality and algorithm parallelism. These methods enhance efficiency in terms of computation graphs, memory management, and training schedules. Since arithmetic equivalence is guaranteed in these proposed methods, we do not sacrifice task-related performance, strictly pushing the performance-efficiency Pareto frontier.

Chapter 4

Conclusions

The machine learning software stack plays a critical role in the machine learning community and serves as the foundation for future artificial intelligence. However, ensuring its efficiency has been and will continue to be a significant challenge. In this dissertation, we propose strategies to address this issue by enhancing the efficiency of the machine learning software stack through algorithm and compilation designs. We summarize our major contributions below.

In Chapter 2, we present efficient machine learning algorithms, highlighting the following key advancements. We improve the efficiency of machine learning datasets, models, and methods.

- We thoroughly investigate the gradient matching method, addressing crucial questions about what, how, and where to match gradients. Our proposal involves matching multi-level gradients, encompassing both intra-class and inter-class gradient information. Furthermore, we demonstrate that the distance function should focus on the angle while considering the magnitude to delay overfitting. Additionally, we introduce an overfitting-aware adaptive learning step strategy to enhance algorithmic efficiency by eliminating unnecessary optimization steps.

- We introduce NormSoftmax, a technique where the input vector is first normalized to unit variance and then processed through the standard softmax function. NormSoftmax effectively resolves the optimization challenges associated with softmax, leading to fast and stable training. We validate the efficacy of NormSoftmax through experiments on Transformer-based models and convolutional neural networks, confirming its effectiveness in stabilizing and speeding up the training process of neural networks utilizing cross-entropy loss or dot-product attention operations.

- We identify limitations in previous work, which often focus on a single layer of the machine learning stack. To overcome this, we propose a new methodology that enables end-to-end joint optimization of neural architecture search and mixed precision quantization. Our approach searches for optimal combinations of architectures and precisions, directly optimizing prediction accuracy and hardware energy consumption. We streamline the process from neural architecture design to hardware deployment, improving efficiency and automation.

In Chapter 3, we present efficient machine learning compilation techniques featuring the following advancements. We improve the efficiency of computation graphs, memory and computation management, and scheduling for training workloads.

- We propose a solution to unify two popular Transformer architectures, Pre-LN and Pre-RMSNorm Transformers. Additionally, we introduce the Pre-CRMSNorm Transformer, which leverages lossless compression on zero-mean vectors. We formally establish the equivalence of Pre-LN, Pre-RMSNorm, and Pre-CRMSNorm Transformer variants in both training and inference, demonstrating that Pre-LN Transformers can

be replaced with Pre-(C)RMSNorm counterparts at a minimal cost. This substitution offers the same arithmetic functionality while providing free efficiency improvements.

- We formulate the decision problem for reversible operators, with training time as the objective function and memory usage as the constraint. By solving this problem, we can maximize training throughput for reversible neural architectures. Our proposed framework automates the decision process, empowering researchers to develop and train reversible neural networks more efficiently.

- We propose optimizer fusion, a technique that combines the optimizer with forward or backward computation. By reordering the forward computation, gradient calculation, and parameter updating, our method improves the efficiency of iterative optimizers. Optimizer fusion, as a general "plug-in" technique, can be applied to the training process without altering the optimizer algorithm, leveraging better hardware locality and computation graph parallelism.

Through the explorations and discussions presented in this dissertation, we have highlighted major challenges in achieving efficiency in the machine learning software stack and provided effective solutions. Nonetheless, there are still opportunities for further improvements, since we have not reached the theoretical computation capabilities of machine learning accelerators. The emergence of new machine learning applications introduces novel efficiency challenges. Therefore, it is worthwhile to delve into the following future research directions and open problems, which hold promise for further exploration.

- Large models, especially large language models, are poised to become the foundation of next-generation data-centric [208] and generative [25] artificial intelligence. However,

training and deploying such massive models remain costly and challenging. Managing large computation resources, including up to 10k accelerators across the data centers [8], is an ongoing challenge that requires further investigation.

- Machine learning workloads extend beyond dense matrix multiplication-based neural networks and backpropagation-based optimization methods. Other categories of machine learning workloads have received less attention but may require additional optimization efforts.

- The mainstream machine learning hardware accelerators are based on Von Neumann architecture and electronic computers. However, emerging architectures and computation platforms, such as quantum computing [205] and optical computing [131], present new opportunities for machine learning applications. Bridging the gap between problem specifications and computation platforms will be essential for the machine learning software stack to evolve accordingly.

Artificial intelligence, particularly machine learning, is widely recognized as a critical area for the next-generation industrial revolution. Improving efficiency in this domain is an ongoing challenge that demands continuous human intelligence and effort.

Appendices

Appendix A

Appendices for Section 2.2

A.1 Proof of Lemmas

A.1.1 Lemma 2.2.1

Let $\boldsymbol{y} = \frac{\boldsymbol{x}}{\sigma(\boldsymbol{x})}, \boldsymbol{z} = \text{softmax}(\boldsymbol{y})$. Following the chain rules, we have

$$\frac{\partial l}{\partial \boldsymbol{y}} = (\text{diag}(\boldsymbol{z}) - \boldsymbol{z}\boldsymbol{z}^T)\frac{\partial l}{\partial \boldsymbol{z}} \tag{A.1}$$

$$\frac{\partial l}{\partial \boldsymbol{x}} = \frac{1}{\sigma(\boldsymbol{x})}\left(\boldsymbol{I} - \frac{\boldsymbol{x}\boldsymbol{x}^T - \mu(\boldsymbol{x})\mathbf{1}\boldsymbol{x}^T}{n\sigma^2(\boldsymbol{x})}\right)\frac{\partial l}{\partial \boldsymbol{y}} = \frac{1}{\sigma(\boldsymbol{x})}\left(\boldsymbol{I} - \frac{\boldsymbol{y}\boldsymbol{y}^T - \mu(\boldsymbol{y})\mathbf{1}\boldsymbol{y}^T}{n}\right)\frac{\partial l}{\partial \boldsymbol{y}} \tag{A.2}$$

With $\mathbf{1}^T\boldsymbol{z} = 1$, we can verify that

$$\mathbf{1}^T\frac{\partial l}{\partial \boldsymbol{y}} = (\mathbf{1}^T\text{diag}(\boldsymbol{z}) - \mathbf{1}^T\boldsymbol{z}\boldsymbol{z}^T)\frac{\partial l}{\partial \boldsymbol{z}} \tag{A.3}$$

$$= (\boldsymbol{z}^T - \boldsymbol{z}^T)\frac{\partial l}{\partial \boldsymbol{z}} \tag{A.4}$$

$$= 0 \tag{A.5}$$

With $n\mu(\boldsymbol{x}) = \mathbf{1}^T\boldsymbol{x}$, we can prove that

$$\mathbf{1}^T\frac{\partial l}{\partial \boldsymbol{x}} = \frac{1}{\sigma(\boldsymbol{x})}\left(\mathbf{1}^T\boldsymbol{I} - \frac{\mathbf{1}^T\boldsymbol{x}\boldsymbol{x}^T - \mu(\boldsymbol{x})\mathbf{1}^T\mathbf{1}\boldsymbol{x}^T}{n\sigma^2(\boldsymbol{x})}\right)\frac{\partial l}{\partial \boldsymbol{y}} \tag{A.6}$$

$$= \frac{1}{\sigma(\boldsymbol{x})}\left(\mathbf{1}^T - \frac{n\mu(\boldsymbol{x})\boldsymbol{x}^T - n\mu(\boldsymbol{x})\boldsymbol{x}^T}{n\sigma^2(\boldsymbol{x})}\right)\frac{\partial l}{\partial \boldsymbol{y}} \tag{A.7}$$

$$= \frac{1}{\sigma(\boldsymbol{x})}\mathbf{1}^T\frac{\partial l}{\partial \boldsymbol{y}} \tag{A.8}$$

$$= 0 \tag{A.9}$$

Hence, we prove that $\mu\left(\frac{\partial l}{\partial \boldsymbol{x}}\right) = \mu\left(\frac{\partial l}{\partial \boldsymbol{y}}\right) = 0$. We then prove the scaling for L^2 norm by leveraging the definition of variance $\sigma^2(\boldsymbol{a}) = \|\boldsymbol{a}\|_2^2/n - \mu^2(\boldsymbol{a})$.

$$\left\|\frac{\partial l}{\partial \boldsymbol{x}}\right\|_2^2 = \left(\frac{\partial l}{\partial \boldsymbol{x}}\right)^T \frac{\partial l}{\partial \boldsymbol{x}} \tag{A.10}$$

$$= \frac{1}{\sigma^2(\boldsymbol{x})}\left(\frac{\partial l}{\partial \boldsymbol{y}}\right)^T \left(\boldsymbol{I} - \frac{\boldsymbol{x}\boldsymbol{x}^T - \mu(\boldsymbol{x})\boldsymbol{x}\boldsymbol{1}^T}{n\sigma^2(\boldsymbol{x})}\right)\left(\boldsymbol{I} - \frac{\boldsymbol{x}\boldsymbol{x}^T - \mu(\boldsymbol{x})\boldsymbol{1}\boldsymbol{x}^T}{n\sigma^2(\boldsymbol{x})}\right)\frac{\partial l}{\partial \boldsymbol{y}} \tag{A.11}$$

$$= \frac{1}{\sigma^2(\boldsymbol{x})}\left(\frac{\partial l}{\partial \boldsymbol{y}}\right)^T \left(\boldsymbol{I} - \frac{\boldsymbol{x}\boldsymbol{x}^T}{n\sigma^2(\boldsymbol{x})} + \frac{\mu(\boldsymbol{x})\boldsymbol{1}\boldsymbol{x}^T + \mu(\boldsymbol{x})\boldsymbol{x}\boldsymbol{1}^T}{n\sigma^2(\boldsymbol{x})}\right)\frac{\partial l}{\partial \boldsymbol{y}} \tag{A.12}$$

$$= \frac{1}{\sigma^2(\boldsymbol{x})}\left(\left\|\frac{\partial l}{\partial \boldsymbol{y}}\right\|_2^2 - \left(\frac{\partial l}{\partial \boldsymbol{y}}\right)^T \frac{\boldsymbol{x}\boldsymbol{x}^T}{n\sigma^2(\boldsymbol{x})}\frac{\partial l}{\partial \boldsymbol{y}} + \frac{\mu(\boldsymbol{x})\left(\frac{\partial l}{\partial \boldsymbol{y}}\right)^T \boldsymbol{1}\boldsymbol{x}^T\frac{\partial l}{\partial \boldsymbol{y}} + \mu(\boldsymbol{x})\left(\frac{\partial l}{\partial \boldsymbol{y}}\right)^T \boldsymbol{x}\boldsymbol{1}^T\frac{\partial l}{\partial \boldsymbol{y}}}{n\sigma^2(\boldsymbol{x})}\right) \tag{A.13}$$

$$= \frac{1}{\sigma^2(\boldsymbol{x})}\left(\left\|\frac{\partial l}{\partial \boldsymbol{y}}\right\|_2^2 - \left(\frac{\partial l}{\partial \boldsymbol{y}}\right)^T \frac{\boldsymbol{x}\boldsymbol{x}^T}{n\sigma^2(\boldsymbol{x})}\frac{\partial l}{\partial \boldsymbol{y}}\right) \tag{A.14}$$

$$= \frac{1}{\sigma^2(\boldsymbol{x})}\left\|\frac{\partial l}{\partial \boldsymbol{y}}\right\|_2^2 - \frac{1}{n\sigma^4(\boldsymbol{x})}\left(\boldsymbol{x}^T\frac{\partial l}{\partial \boldsymbol{y}}\right)^2 \tag{A.15}$$

$$\leq \frac{1}{\sigma^2(\boldsymbol{x})}\left\|\frac{\partial l}{\partial \boldsymbol{y}}\right\|_2^2 \tag{A.16}$$

From the definition of variance, we obtain that

$$\sigma^2\left(\frac{\partial l}{\partial \boldsymbol{x}}\right) = \left\|\frac{\partial l}{\partial \boldsymbol{x}}\right\|_2^2/n \leq \frac{1}{\sigma^2(\boldsymbol{x})}\left\|\frac{\partial l}{\partial \boldsymbol{y}}\right\|_2^2/n = \frac{1}{\sigma^2(\boldsymbol{x})}\sigma^2\left(\frac{\partial l}{\partial \boldsymbol{y}}\right) \tag{A.17}$$

Above all, we prove that

$$\left\|\frac{\partial l}{\partial \boldsymbol{x}}\right\|_2 \leq \frac{\left\|\frac{\partial l}{\partial \boldsymbol{y}}\right\|_2}{\sigma(\boldsymbol{x})}, \sigma\left(\frac{\partial l}{\partial \boldsymbol{x}}\right) \leq \frac{\sigma\left(\frac{\partial l}{\partial \boldsymbol{y}}\right)}{\sigma(\boldsymbol{x})} \tag{A.18}$$

A.1.2 Lemma 2.2.2

Given $\boldsymbol{x}_2 = k\boldsymbol{x}_1$, we have $\boldsymbol{x}_1/\sigma(\boldsymbol{x}_1) = \boldsymbol{x}_2/\sigma(\boldsymbol{x}_2)$. Thus, we obtain that $\boldsymbol{z}_1 = \text{softmax}(\boldsymbol{x}_1/\sigma(\boldsymbol{x}_1)) = \boldsymbol{z}_2 = \text{softmax}(\boldsymbol{x}_2/\sigma(\boldsymbol{x}_2))$. Now that $\boldsymbol{z}_1 = \boldsymbol{z}_2$, we can obtain the

same loss $f(\boldsymbol{z}_1) = f(\boldsymbol{z}_2)$ and generate the same gradient $\frac{\partial l_1}{\partial \boldsymbol{z}_1} = \frac{\partial l_2}{\partial \boldsymbol{z}_2}$. Based on Equation equation A.2, we conclude that $\frac{\partial l_2}{\partial \boldsymbol{x}_2} = \frac{1}{k}\frac{\partial l_1}{\partial \boldsymbol{x}_1}$.

A.2 Experimental Details

A.2.1 Vision Transformers on CIFAR10

We train from scratch with AdamW optimizer for 100 epochs (50,000 iterations with 100 mini-batch size) in mixed precision. The resolution of an input image is $3 \times 32 \times 32$, and the patch size is 4. The hidden size, MLP size, number of heads, dimension of heads, and number of layers are 256, 1024, 8, 32, and 8, respectively. We use the following attention function,

$$\text{Attention}(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}) = \text{softmax}\left(\frac{\boldsymbol{Q}\boldsymbol{K}^T}{\min(\sigma(\boldsymbol{Q}\boldsymbol{K}^T), \gamma)}\right)\boldsymbol{V} \tag{A.19}$$

where the softmax and the standard deviation are calculated along the same axes. We set $\gamma = \sqrt{d}$ and $+\infty$. The learning rate is 1e-3, and a strong weight decay 1e-1 is applied. The learning rate linearly increases from 2e-4 with 5 warmup epochs and then decays to 0 with the cosine annealing scheduler. We also enable label smoothing (0.1), random erasing with the probability of 0.1, mixup with $\alpha = 0.2$, cutmix with $\alpha = 1.0$, and TrivialAugment. The code is attached in the supplementary material.

A.2.2 Vision Transformers on ImageNet

We follow the torchvision's reference implementation. The batch size is $512 \times 8 = 4096$. We train from scratch with AdamW optimizer in mixed precision. The learning rate is 3e-3, and the weight decay is 3e-1. The learning rate is linearly increased from $3e-3 \times 0.033$ with 30 warmup epochs and then decays to 0 with the cosine annealing scheduler. We also enable

label smoothing (0.1), mixup with $\alpha = 0.2$, cutmix with $\alpha = 1.0$, clipping gradient norm with 1, RandAugment [38], repeated augmentation with 3 repetitions, exponential moving average for model parameters.

A.2.3 ResNet on ImageNet

We follow the example in the JAX framework. We only enable horizontal flip data augmentation. We use SGDM with a mini-batch size of 8,192, a learning rate of 3.2, a momentum of 0.9, and a weight decay of 1e-4. The learning rate is warmup in 5 epochs.

A.2.4 Machine Translation

We use exactly the same settings as [202]. Please refer to the original paper for reference.

A.3 Cost Analysis of NormSoftmax

Since we import normalization in the softmax, we inevitably introduce the extra computation and memory cost of NormSoftmax. However, we show that the overhead of the proposed lightweight NormSoftmax is negligible for large machine learning models.

Memory cost. Equation 2.21 indicates that we can preprocess the input vector x and pass it to NormSoftmax. We can fuse the γ factor with the ascending layer of NormSoftmax if possible during inference. For a linear layer $\boldsymbol{x} = \boldsymbol{A}\boldsymbol{y} + \boldsymbol{b}$, we can always fuse the γ with $\boldsymbol{A}$ and $\boldsymbol{b}$, as we do not need to save γ during inference. For training, we need to pay the extra memory cost for normalization since we have to save intermediate results for gradient computation.

Computation cost. For training, we have to pay the cost of calculating the variance and the corresponding gradients. For inference, we may discard the normalization if the softmax is in the last layer. In a classification model with cross entropy loss, we can discard the normalization since the normalization has no impact on the classification result.

In our experiments on Transformer, NormSoftmax induces $0.2\% - 1\%$ extra execution time. For the experiments for cross-entropy loss, the extra execution time is less than 0.05% since we only add one normalization layer.

Appendix B

Appendices for Section 3.1

B.1 Proof for Lemma 3.1.1

Given a linear transformation $\boldsymbol{y} = \boldsymbol{A}\boldsymbol{x} + \boldsymbol{b}, \boldsymbol{x} \in \mathbb{R}^n, \boldsymbol{A} \in \mathbb{R}^{m\times n}, \boldsymbol{b}, \boldsymbol{y} \in \mathbb{R}^m$, we have

$$\begin{aligned}
\boldsymbol{y} &= \boldsymbol{A}\boldsymbol{x} + \boldsymbol{b} && \text{(B.1)}\\
&= (\boldsymbol{A} - k\mathbf{1}\mathbf{1}^T\boldsymbol{A})\boldsymbol{x} + k\mathbf{1}\mathbf{1}^T\boldsymbol{A}\boldsymbol{x} + (\boldsymbol{b} - \mu(\boldsymbol{b})\mathbf{1}) + \mu(\boldsymbol{b})\mathbf{1} && \text{(B.2)}\\
&= (\boldsymbol{A} - k\mathbf{1}\mathbf{1}^T\boldsymbol{A})\boldsymbol{x} + (\boldsymbol{b} - \mu(\boldsymbol{b})\mathbf{1}) + k(\mathbf{1}^T\boldsymbol{A}\boldsymbol{x})\mathbf{1} + \mu(\boldsymbol{b})\mathbf{1} && \text{(B.3)}\\
&= (\boldsymbol{A} - k\mathbf{1}\mathbf{1}^T\boldsymbol{A})\boldsymbol{x} + (\boldsymbol{b} - \mu(\boldsymbol{b})\mathbf{1}) + (k\mathbf{1}^T\boldsymbol{A}\boldsymbol{x} + \mu(\boldsymbol{b}))\mathbf{1} && \text{(B.4)}\\
&= \hat{\boldsymbol{A}}\boldsymbol{x} + \hat{\boldsymbol{b}} + f(\boldsymbol{x}, k)\mathbf{1} && \text{(B.5)}
\end{aligned}$$

where $\hat{\boldsymbol{A}} = \boldsymbol{A} - k\mathbf{1}\mathbf{1}^T\boldsymbol{A}, \hat{\boldsymbol{b}} = \boldsymbol{b} - \mu(\boldsymbol{b})\mathbf{1}, f(\boldsymbol{x}, k) = k\mathbf{1}^T\boldsymbol{A}\boldsymbol{x} + \mu(\boldsymbol{b})$.

If $k = 1/m$, then we obtain

$$\begin{aligned}
\mu(\hat{\boldsymbol{A}}\boldsymbol{x}) &= \frac{1}{m}\mathbf{1}^T(\boldsymbol{A} - \frac{1}{m}\mathbf{1}\mathbf{1}^T\boldsymbol{A})\boldsymbol{x} = \frac{1}{m}(\mathbf{1}^T\boldsymbol{A} - \mathbf{1}^T\boldsymbol{A})\boldsymbol{x} = 0 && \text{(B.6)}\\
\mu(\boldsymbol{y}) &= \mu(\hat{\boldsymbol{A}}\boldsymbol{x}) + \mu(\hat{\boldsymbol{b}}) + \mu(f(\boldsymbol{x}, k = 1/m)\mathbf{1}) && \text{(B.7)}\\
&= 0 + 0 + f(\boldsymbol{x}, k = 1/m) && \text{(B.8)}\\
&= \frac{1}{m}\mathbf{1}^T\boldsymbol{A}\boldsymbol{x} + \mu(\boldsymbol{b}) && \text{(B.9)}
\end{aligned}$$

The $\hat{\boldsymbol{A}}$ is the recentered matrix of $\boldsymbol{A}$, and all its column vectors have zero-mean. We decompose the output into two parts.

- The first part $\hat{\boldsymbol{A}}\boldsymbol{x} + \hat{\boldsymbol{b}} = \boldsymbol{y} - \mu(\boldsymbol{y})\mathbf{1}$, with zero mean, is another linear transformation with $\hat{\boldsymbol{A}} = \boldsymbol{A} - \frac{1}{m}\mathbf{1}\mathbf{1}^T\boldsymbol{A}, \hat{\boldsymbol{b}} = \boldsymbol{b} - \mu(\boldsymbol{b})\mathbf{1}$.
- The second part corresponds to the mean information $\mu(\boldsymbol{y})\mathbf{1} = (\frac{1}{m}\mathbf{1}^T\boldsymbol{A}\boldsymbol{x} + \mu(\boldsymbol{b}))\mathbf{1}$.

B.2 Post-LN Transformers

Different from the Pre-LN Transformers, the Post-LN Transformers have the following blocks.

$$\boldsymbol{x}_{l+1} = \mathrm{LN}(\boldsymbol{x}_l + \mathcal{F}_l(\boldsymbol{x}_l)),\ l = 0, 1, ..., L-1, \tag{B.10}$$

Layer normalization is on the main branch instead of the beginning of residual branches. We can keep a zero-mean branch on the main branch without impact on the functionality.

$$\begin{aligned}
\boldsymbol{x}_{l+1} &= \mathrm{LN}(\boldsymbol{x}_l + \mathcal{F}_l(\boldsymbol{x}_l)) && \text{(B.11)}\\
&= \mathrm{LN}((\boldsymbol{x}_l - \mu(\boldsymbol{x}_l)\mathbf{1}) + (\mathcal{F}_l(\boldsymbol{x}_l) - \mu(\mathcal{F}_l(\boldsymbol{x}_l))\mathbf{1})) && \text{(B.12)}\\
&= \mathrm{LN}(\hat{\boldsymbol{x}}_l + \hat{\mathcal{F}}_l(\boldsymbol{x}_l)) && \text{(B.13)}\\
&= \mathrm{RMSNorm}(\hat{\boldsymbol{x}}_l + \hat{\mathcal{F}}_l(\boldsymbol{x}_l)) && \text{(B.14)}
\end{aligned}$$

For the residual branch $\mathcal{F}_l$, we can apply the same method in Pre-LN Transformer. We can modify the output linear projection to obtain $\hat{\mathcal{F}}_l$, which will generate the zero-mean part of the original result.

The recentering operation $\hat{\boldsymbol{x}}_l = \boldsymbol{x}_l - \mu(\boldsymbol{x}_l)\mathbf{1}$ requires extra computation. If elementwise affine transformation is disabled in LayerNorm, $\boldsymbol{x}_l$ is the output of a normalization such that $\mu(\boldsymbol{x}_l) = 0$ and $\hat{\boldsymbol{x}}_l = \boldsymbol{x}_l$. If the transformation is enabled, $\boldsymbol{x}_l$ is not guaranteed zero-mean such that the explicit recentering is necessary.

B.3 Experiments

B.3.1 Implementation of Normalization

We have provided our implementation with JAX and PyTorch in the supplementary material. The reported results are based on the following implementations.

For JAX, we use the APIs of LayerNorm and RMSNorm in the flax library. For PyTorch, we use the implementations of LayerNorm and RMSNorm from NVIDIA's apex extension [1]. For CRMSNorm, we use our own customized implementations. We also provide our customized implementation of LayerNorm, RMSNorm.

We notice that there are lots of APIs for the standard LayerNorm and RMSNorm. For example, PyTorch has provided the official LayerNorm API but lacks of RMSNorm implementation. These different implementations are mixed. We do not find one implementation that is dominant over others for all the cases. For instance, `torch.nn.LayerNorm` is usually faster than apex's one when the input vectors are small in inference, while it is slower than the apex's one when the input vectors are large. PyTorch's official implementation is also slower than apex's for training.

B.3.2 Extended Experiments in ViT

Table B.1 list the architecture parameters of Vision Transformer. We first measure the **inference time**. We sweep these 6 ViTs with 6 batch sizes (1, 4, 16, 64, 256, 1024) and collect the medians of these 36 data points. We report the average of these 36 experiments in Table B.2. We conduct inference on a single A100 with automatic mixed precision (amp)

[1] https://github.com/NVIDIA/apex

Name	Dimension	Depth	Heads	MLP Dimension
Tiny-16	192	12	3	192×4
Small-16	384	12	6	384×4
Base-16	768	12	12	768×4
Large-16	1024	24	16	1024×4
Huge-14	1280	32	16	1280×4
Giant-14	1664	48	16	8192

Table B.1: ViTs with different sizes. The number in the model name is the patch size.

	no norm	**Pre-LN**	**Pre-RMS**	**Pre-CRMS**
PyTorch, single A100, amp	0.8567	1.000	0.9699	0.9783
amp $\rightarrow$ float32	0.9353	1.000	0.9850	0.9951
single A100 $\rightarrow$ 16-thread CPU	0.8697	1.000	0.9012	0.8857
PyTorch $\rightarrow$ JAX	0.9610	1.000	0.9873	1.0005

Table B.2: Normalized inference time of ViT.

in PyTorch. We further change the precision (disabling the amp), computation platforms (16 threads in AMD EPYC 7742 CPUs), and machine learning frameworks (JAX).

B.3.3 Numerical Issue

The theoretical arithmetic equivalence cannot be fully translated into equality in the practical numerical computation if we use floating numbers. An intuitive example is that $\mu(\boldsymbol{x}+\boldsymbol{y}) = \mu(\boldsymbol{x}) + \mu(vy)$ always holds for any vectors $\boldsymbol{x}, \boldsymbol{y}$. However, if these two vectors are represented as (low precision) floating numbers, this equality is not guaranteed in real-world numerical computation. It is possible that these small discrepancies may be accumulated and enlarged in the large models, further degrading the numerical stability. In our proposed method, we cannot ensure the exactly zero-mean in the main branch numerically.

The numerical issue is a common problem in machine learning. A typical example

is operator reordering and layer fusion. PyTorch provides a related API officially, named `torch.ao.quantization.fuse_modules`. We can fuse the convolution layer and its following batch normalization layer to simplify the computation. These two layers are separate in training and can be fused to accelerate the inference. The fusion does not break the arithmetic equivalence but changes the numerical results. In spite of the numerical difference, the fusion usually has a neutral impact on task-related performance, such as classification accuracy, even in large models. Fine-tuning or calibration may be helpful in case there is severe performance degradation.

Our proposed methods encounter a similar issue as layer fusion since we modify partial parameters. In our experiments, we can convert the pre-trained Pre-LN ViT-H/14 into Pre-(C)RMS variants without any accuracy change on the ImageNet validation dataset. Actually, we observe that replacing PyTorch's official LayerNorm implementation with the apex's one may have a larger impact on the model performance.

Bibliography

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems, 2015. Software available from tensorflow.org.

[2] Atish Agarwala, Samuel Stern Schoenholz, Jeffrey Pennington, and Yann N. Dauphin. Temperature Check: Theory and Practice for Training Models with Softmax-Cross-Entropy Losses. *Transactions on Machine Learning Research (TMLR)*, 2023.

[3] Rahaf Aljundi, Min Lin, Baptiste Goujaud, and Yoshua Bengio. Gradient Based Sample Selection for Online Continual Learning. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 32. Curran Associates, Inc., 2019.

[4] Manoj Alwani, Han Chen, Michael Ferdman, and Peter Milder. Fused-Layer CNN Accelerators. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE Press, 2016.

[5] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. Layer Normalization. *arXiv preprint arXiv:1607.06450*, 2016.

[6] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural Machine Translation by Jointly Learning to Align and Translate. *International Conference on Learning Representations (ICLR)*, 2015.

[7] Ron Banner, Itay Hubara, Elad Hoffer, and Daniel Soudry. Scalable Methods for 8-bit Training of Neural Networks. *Advances in Neural Information Processing Systems (NeurIPS)*, 31, 2018.

[8] Paul Barham, Aakanksha Chowdhery, Jeff Dean, Sanjay Ghemawat, Steven Hand, Daniel Hurt, Michael Isard, Hyeontaek Lim, Ruoming Pang, Sudip Roy, et al. Pathways: Asynchronous Distributed Dataflow for ML. *Proceedings of Machine Learning and Systems (MLSys)*, 4:430–449, 2022.

[9] E. Belouadah and A. Popescu. ScaIL: Classifier Weights Scaling for Class Incremental Learning. In *IEEE Winter Conference on Applications of Computer Vision (WACV)*, pages 1255–1264, Los Alamitos, CA, USA, Mar 2020. IEEE Computer Society.

[10] Yoshua Bengio and Yann LeCun. Scaling Learning Algorithms Towards AI. In *Large Scale Kernel Machines*. MIT Press, 2007.

[11] James Bergstra, Daniel Yamins, and David Cox. Making a Science of Model Search: Hyperparameter Optimization in Hundreds of Dimensions for Vision Architectures. In *International Conference on Machine Learning (ICML)*, pages 115–123. PMLR, 2013.

[12] David Berthelot, Nicholas Carlini, Ekin D Cubuk, Alex Kurakin, Kihyuk Sohn, Han Zhang, and Colin Raffel. ReMixMatch: Semi-Supervised Learning with Distribution Alignment and Augmentation Anchoring. *International Conference on Learning Representations (ICLR)*, 2020.

[13] Nils Bjorck, Carla P Gomes, Bart Selman, and Kilian Q Weinberger. Understanding Batch Normalization. *Advances in Neural Information Processing Systems (NeurIPS)*, 31, 2018.

[14] Stefano B Blumberg, Ryutaro Tanno, Iasonas Kokkinos, and Daniel C Alexander. Deeper Image Quality Transfer: Training Low-Memory Neural Networks for 3D Images. In *Medical Image Computing and Computer Assisted Intervention (MICCAI)*, pages 118–125. Springer, 2018.

[15] Daniel Bolya, Cheng-Yang Fu, Xiaoliang Dai, Peizhao Zhang, and Judy Hoffman. Hydra Attention: Efficient Attention with Many Heads. In Leonid Karlinsky, Tomer Michaeli, and Ko Nishino, editors, *European Conference on Computer Vision (ECCV) Workshops*, pages 35–49, Cham, 2023. Springer Nature Switzerland.

[16] Rishi Bommasani, Drew A Hudson, Ehsan Adeli, Russ Altman, Simran Arora, Sydney von Arx, Michael S Bernstein, Jeannette Bohg, Antoine Bosselut, Emma Brunskill, et al. On the Opportunities and Risks of Foundation Models. *arXiv preprint arXiv:2108.07258*, 2021.

[17] Yelysei Bondarenko, Markus Nagel, and Tijmen Blankevoort. Understanding and Overcoming the Challenges of Efficient Transformer Quantization. In *Proceedings*

of the Conference on Empirical Methods in Natural Language Processing (EMNLP), November 2021.

[18] Zalán Borsos, Mojmir Mutny, and Andreas Krause. Coresets via Bilevel Optimization for Continual Learning and Streaming. In H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems (NeurIPS)*, volume 33, pages 14879–14890. Curran Associates, Inc., 2020.

[19] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: composable transformations of Python+NumPy programs, 2018.

[20] John Bridle. Training Stochastic Model Recognition Algorithms as Networks can Lead to Maximum Mutual Information Estimation of Parameters. In D. Touretzky, editor, *Advances in Neural Information Processing Systems (NeurIPS)*, volume 2. Morgan-Kaufmann, 1989.

[21] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language Models Are Few-Shot Learners. *Advances in Neural Information Processing Systems (NeurIPS)*, 33:1877–1901, 2020.

[22] Robin Brügger, Christian F. Baumgartner, and Ender Konukoglu. A Partially Reversible U-Net for Memory-Efficient Volumetric Image Segmentation. *Medical Image Computing and Computer Assisted Intervention (MICCAI)*, page 429–437, 2019.

[23] S. R. Bulò, L. Porzi, and P. Kontschieder. In-place Activated BatchNorm for Memory-Optimized Training of DNNs. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 5639–5647, June 2018.

[24] Han Cai, Ligeng Zhu, and Song Han. ProxylessNAS: Direct Neural Architecture Search on Target Task and Hardware. *International Conference on Learning Representations (ICLR)*, 2019.

[25] Yihan Cao, Siyu Li, Yixin Liu, Zhiling Yan, Yutong Dai, Philip S Yu, and Lichao Sun. A Comprehensive Survey of AI-Generated Content (AIGC): A History of Generative AI from GAN to ChatGPT. *arXiv preprint arXiv:2303.04226*, 2023.

[26] Francisco M Castro, Manuel J Marín-Jiménez, Nicolás Guil, Cordelia Schmid, and Karteek Alahari. End-to-End Incremental Learning. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 233–248, 2018.

[27] Mauro Cettolo, Jan Niehues, Sebastian Stüker, Luisa Bentivogli, Roldano Cattoni, and Marcello Federico. The IWSLT 2015 Evaluation Campaign. In *Proceedings of the 12th International Workshop on Spoken Language Translation: Evaluation Campaign*, pages 2–14, Da Nang, Vietnam, December 3-4 2015.

[28] Junkun Chen, Kaiyu Chen, Xinchi Chen, Xipeng Qiu, and Xuanjing Huang. Exploring Shared Structures and Hierarchies for Multiple NLP Tasks. *arXiv preprint arXiv:1808.07658*, 2018.

[29] Lili Chen, Kevin Lu, Aravind Rajeswaran, Kimin Lee, Aditya Grover, Misha Laskin, Pieter Abbeel, Aravind Srinivas, and Igor Mordatch. Decision Transformer: Reinforce-

ment Learning via Sequence Modeling. *Advances in Neural Information Processing Systems (NeurIPS)*, 34:15084–15097, 2021.

[30] Tian Qi Chen, Yulia Rubanova, Jesse Bettencourt, and David K Duvenaud. Neural Ordinary Differential Equations. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems (NeurIPS)*, pages 6571–6583. Curran Associates, Inc., 2018.

[31] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. *CoRR*, abs/1512.01274, 2015.

[32] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training Deep Nets with Sublinear Memory Cost. *arXiv preprint arXiv:1604.06174*, 2016.

[33] Xiangning Chen, Cho-Jui Hsieh, and Boqing Gong. When Vision Transformers Outperform ResNets without Pre-training or Strong Data Augmentations. *International Conference on Learning Representations (ICLR)*, 2022.

[34] Yutian Chen, Max Welling, and Alex Smola. Super-Samples from Kernel Herding. In *Proceedings of the 26th Conference on Uncertainty in Artificial Intelligence (UAI)*, page 109–116, Arlington, Virginia, USA, 2010. AUAI Press.

[35] Vitaliy Chiley, Vithursan Thangarasa, Abhay Gupta, Anshul Samar, Joel Hestness, and Dennis DeCoste. RevBiFPN: The Fully Reversible Bidirectional Feature Pyramid Network. *Proceedings of Machine Learning and Systems (MLSys)*, 5, 2023.

[36] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. PaLM: Scaling Language Modeling with Pathways. *arXiv preprint arXiv:2204.02311*, 2022.

[37] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. BinaryConnect: Training Deep Neural Networks with binary weights during propagations. *Advances in Neural Information Processing Systems (NeurIPS)*, 28, 2015.

[38] Ekin D Cubuk, Barret Zoph, Jonathon Shlens, and Quoc V Le. RandAugment: Practical automated data augmentation with a reduced search space. In *Proceedings of the IEEE/CVF conference on Computer Vision and Pattern Recognition (CVPR) workshops*, pages 702–703, 2020.

[39] Justin Cui, Ruochen Wang, Si Si, and Cho-Jui Hsieh. DC-BENCH: Dataset Condensation Benchmark. *Advances in Neural Information Processing Systems (NeurIPS)*, 2022.

[40] Yann Le Cun, John S. Denker, and Sara A. Solla. *Optimal Brain Damage*, page 598–605. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1990.

[41] Raj Dabre, Chenhui Chu, and Anoop Kunchukuttan. A Survey of Multilingual Neural Machine Translation. *ACM Computing Surveys*, 53(5):1–38, 2020.

[42] Mostafa Dehghani, Josip Djolonga, Basil Mustafa, Piotr Padlewski, Jonathan Heek, Justin Gilmer, Andreas Steiner, Mathilde Caron, Robert Geirhos, Ibrahim Alabdul-

mohsin, et al. Scaling vision transformers to 22 billion parameters. *arXiv preprint arXiv:2302.05442*, 2023.

[43] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. ImageNet: A large-scale hierarchical image database. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 248–255, 2009.

[44] Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. LLM.int8(): 8-bit Matrix Multiplication for Transformers at Scale. *Advances in Neural Information Processing Systems (NeurIPS)*, 2022.

[45] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT)*, pages 4171–4186, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics.

[46] Caiwen Ding, Siyu Liao, Yanzhi Wang, Zhe Li, Ning Liu, Youwei Zhuo, Chao Wang, Xuehai Qian, Yu Bai, Geng Yuan, Xiaolong Ma, Yipeng Zhang, Jian Tang, Qinru Qiu, Xue Lin, and Bo Yuan. CirCNN: Accelerating and Compressing Deep Neural Networks Using Block-Circulant Weight Matrices. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, page 395–408, New York, NY, USA, 2017. Association for Computing Machinery.

[47] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold,

Sylvain Gelly, et al. An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale. *International Conference on Learning Representations (ICLR)*, 2021.

[48] John Duchi, Elad Hazan, and Yoram Singer. Adaptive Subgradient Methods for Online Learning and Stochastic Optimization. *Journal of Machine Learning Research (JMLR)*, 12:2121–2159, July 2011.

[49] Thomas Elsken, Jan Hendrik Metzen, Frank Hutter, et al. Neural Architecture Search: A Survey. *Journal of Machine Learning Research (JMLR)*, 20(55):1–21, 2019.

[50] Massimiliano Fatica. CUDA toolkit and libraries. In *2008 IEEE Hot Chips 20 Symposium (HCS)*, pages 1–22, 2008.

[51] Dan Feldman, Melanie Schmidt, and Christian Sohler. Turning Big Data into Tiny Data: Constant-Size Coresets for k-Means, PCA and Projective Clustering. In *Proceedings of the 24th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, page 1434–1453, USA, 2013. Society for Industrial and Applied Mathematics.

[52] Matthias Feurer and Frank Hutter. Hyperparameter Optimization. In *Automated machine learning*, pages 3–33. Springer, Cham, 2019.

[53] Pierre Foret, Ariel Kleiner, Hossein Mobahi, and Behnam Neyshabur. Sharpness-Aware Minimization for Efficiently Improving Generalization. *International Conference on Learning Representations (ICLR)*, 2021.

[54] Zongxion Geng, Jiahui andg Chen, Yuandou Wang, Herbert Woisetschlaeger, Sonja Schimmler, Ruben Mayer, Zhiming Zhao, and Chunming Rong. A Survey on Dataset

Distillation: Approaches, Applications and Future Directions. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, 2023.

[55] Spyros Gidaris and Nikos Komodakis. Dynamic Few-Shot Visual Learning without Forgetting. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 4367–4375, 2018.

[56] Aidan N Gomez, Mengye Ren, Raquel Urtasun, and Roger B Grosse. The reversible residual network: Backpropagation without storing activations. *Advances in Neural Information Processing Systems (NeurIPS)*, 30, 2017.

[57] Chengyue Gong, Zixuan Jiang, Dilin Wang, Yibo Lin, Qiang Liu, and David Z. Pan. Mixed Precision Neural Architecture Search for Energy Efficient Deep Learning. In *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–7, 2019.

[58] Ian Goodfellow, Yoshua Bengio, Aaron Courville, and Yoshua Bengio. *Deep Learning*, volume 1. MIT Press, 2016.

[59] Akhilesh Gotmare, Nitish Shirish Keskar, Caiming Xiong, and Richard Socher. A Closer Look at Deep Learning Heuristics: Learning rate restarts, Warmup and Distillation. *International Conference on Learning Representations (ICLR)*, 2019.

[60] Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. Serving DNNs like Clockwork: Performance Predictability from the Bottom Up. In *14th USENIX Symposium on Operating Systems*

Design and Implementation (OSDI), pages 443–462. USENIX Association, November 2020.

[61] Zichao Guo, Xiangyu Zhang, Haoyuan Mu, Wen Heng, Zechun Liu, Yichen Wei, and Jian Sun. Single Path One-Shot Neural Architecture Search with Uniform Sampling. In *European Conference on Computer Vision (ECCV)*, pages 544–560. Springer, 2020.

[62] Song Han, Huizi Mao, and William J Dally. Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding. *International Conference on Learning Representations (ICLR)*, 2016.

[63] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016.

[64] Tong He, Zhi Zhang, Hang Zhang, Zhongyue Zhang, Junyuan Xie, and Mu Li. Bag of Tricks for Image Classification with Convolutional Neural Networks. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition (CVPR)*, pages 558–567, 2019.

[65] Dan Hendrycks and Kevin Gimpel. Gaussian Error Linear Units (GELUs). *arXiv preprint arXiv:1606.08415*, 2016.

[66] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the Knowledge in a Neural Network. *arXiv preprint arXiv:1503.02531*, 2015.

[67] Geoffrey E. Hinton, Simon Osindero, and Yee Whye Teh. A Fast Learning Algorithm for Deep Belief Nets. *Neural Computation*, 18:1527–1554, 2006.

[68] Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, et al. An Empirical Analysis of Compute-Optimal Large Language Model Training. *Advances in Neural Information Processing Systems (NeurIPS)*, 35:30016–30030, 2022.

[69] Youngkyu Hong, Seungju Han, Kwanghee Choi, Seokjun Seo, Beomsu Kim, and Buru Chang. Disentangling Label Distribution for Long-tailed Visual Recognition. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 6626–6636, 2021.

[70] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. *arXiv preprint arXiv:1704.04861*, 2017.

[71] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. Densely Connected Convolutional Networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR)*, pages 4700–4708, 2017.

[72] Gao Huang, Yu Sun, Zhuang Liu, Daniel Sedra, and Kilian Q Weinberger. Deep Networks with Stochastic Depth. In *European Conference on Computer Vision (ECCV)*, pages 646–661. Springer, 2016.

[73] Lei Huang, Jie Qin, Yi Zhou, Fan Zhu, Li Liu, and Ling Shao. Normalization Techniques in Training DNNs: Methodology, Analysis and Application. *IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI)*, 2023.

[74] Xiao Shi Huang, Felipe Perez, Jimmy Ba, and Maksims Volkovs. Improving Transformer Optimization Through Better Initialization. In Hal Daumé III and Aarti Singh, editors, *Proceedings of the 37th International Conference on Machine Learning (ICML)*, volume 119 of *Proceedings of Machine Learning Research*, pages 4475–4483. PMLR, 13–18 Jul 2020.

[75] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Quantized Neural Networks: Training Neural Networks with Low Precision Weights and Activations. *The Journal of Machine Learning Research (JMLR)*, 18(1):6869–6898, 2017.

[76] Sergey Ioffe and Christian Szegedy. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. In *Proceedings of the 32nd International Conference on International Conference on Machine Learning (ICML)*, page 448–456. JMLR.org, 2015.

[77] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry 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 (CVPR)*, pages 2704–2713, 2018.

[78] Jörn-Henrik Jacobsen, Arnold W.M. Smeulders, and Edouard Oyallon. i-RevNet: Deep Invertible Networks. In *International Conference on Learning Representations (ICLR)*, 2018.

[79] Paras Jain, Ajay Jain, Aniruddha Nrusimha, Amir Gholami, Pieter Abbeel, Joseph Gonzalez, Kurt Keutzer, and Ion Stoica. Checkmate: Breaking the Memory Wall with Optimal Tensor Rematerialization. *Proceedings of Machine Learning and Systems (MLSys)*, 2:497–511, 2020.

[80] Eric Jang, Shixiang Gu, and Ben Poole. Categorical Reparameterization with Gumbel-Softmax. *International Conference on Learning Representations (ICLR)*, 2017.

[81] Zhihao Jia, Sina Lin, Charles R. Qi, and Alex Aiken. Exploring Hidden Dimensions in Accelerating Convolutional Neural Networks. In Jennifer Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning (ICML)*, volume 80 of *Proceedings of Machine Learning Research*, pages 2274–2283, Stockholmsmässan, Stockholm Sweden, 10–15 Jul 2018. PMLR.

[82] Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. TASO: Optimizing Deep Learning Computation with Automatic Generation of Graph Substitutions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, page 47–62, New York, NY, USA, 2019. Association for Computing Machinery.

[83] Zixuan Jiang, Jiaqi Gu, Mingjie Liu, and David Z Pan. Delving into Effective Gradient Matching for Dataset Condensation. *IEEE International Conference on Omni-layer Intelligent Systems (COINS)*, 2023.

[84] Zixuan Jiang, Jiaqi Gu, Mingjie Liu, Keren Zhu, and David Z Pan. Optimizer Fusion: Efficient Training with Better Locality and Parallelism. *Hardware Aware Efficient*

Training (HAET) workshop, International Conference on Learning Representations (ICLR), 2021.

[85] Zixuan Jiang, Jiaqi Gu, and David Z Pan. NormSoftmax: Normalizing the Input of Softmax to Accelerate and Stabilize Training. *IEEE International Conference on Omni-layer Intelligent Systems (COINS)*, 2023.

[86] Zixuan Jiang, Jiaqi Gu, Hanqing Zhu, and David Z Pan. Pre-RMSNorm and Pre-CRMSNorm Transformers: Equivalent and Efficient Pre-LN Transformers. *arXiv preprint arXiv:2305.14858*, 2023.

[87] Zixuan Jiang, Keren Zhu, Mingjie Liu, Jiaqi Gu, and David Z. Pan. An Efficient Training Framework for Reversible Neural Architectures. In Andrea Vedaldi, Horst Bischof, Thomas Brox, and Jan-Michael Frahm, editors, *European Conference on Computer Vision (ECCV)*, pages 275–289, Cham, 2020. Springer International Publishing.

[88] Michael I Jordan and Tom M Mitchell. Machine learning: Trends, Perspectives, and Prospects. *Science*, 349(6245):255–260, 2015.

[89] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon,

James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA)*, page 1–12, New York, NY, USA, 2017. Association for Computing Machinery.

[90] Rakhee Kallimani, Krishna Pai, Prasoon Raghuwanshi, Sridhar Iyer, and Onel LA López. TinyML: Tools, Applications, Challenges, and Future Research Directions. *arXiv preprint arXiv:2303.13569*, 2023.

[91] Salman Khan, Muzammal Naseer, Munawar Hayat, Syed Waqas Zamir, Fahad Shahbaz Khan, and Mubarak Shah. Transformers in Vision: A Survey. *ACM Computing Surveys*, 2021.

[92] Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. In *International Conference on Learning Representations (ICLR)*, 2015.

[93] Durk P Kingma and Prafulla Dhariwal. Glow: Generative Flow with Invertible 1x1 Convolutions. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems (NeurIPS)*, pages 10215–10224. Curran Associates, Inc., 2018.

[94] James Kirkpatrick, Razvan Pascanu, Neil Rabinowitz, Joel Veness, Guillaume Desjardins, Andrei A Rusu, Kieran Milan, John Quan, Tiago Ramalho, Agnieszka Grabska-Barwinska, et al. Overcoming Catastrophic Forgetting in Neural Networks. *Proceedings of the National Academy of Sciences (PNAS)*, 114(13):3521–3526, 2017.

[95] Nikita Kitaev, Łukasz Kaiser, and Anselm Levskaya. Reformer: The Efficient Transformer. *International Conference on Learning Representations (ICLR)*, 2020.

[96] Simon Kornblith, Ting Chen, Honglak Lee, and Mohammad Norouzi. Why Do Better Loss Functions Lead to Less Transferable Features? *Advances in Neural Information Processing Systems (NeurIPS)*, 34:28648–28662, 2021.

[97] Raghuraman Krishnamoorthi. Quantizing deep convolutional networks for efficient inference: A whitepaper. *arXiv preprint arXiv:1806.08342*, 2018.

[98] Alex Krizhevsky, Geoffrey Hinton, et al. Learning Multiple Layers of Features from Tiny Images. 2009.

[99] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. ImageNet Classification with Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems (NeurIPS)*, page 1097–1105, Red Hook, NY, USA, 2012. Curran Associates Inc.

[100] HT Kung and Charles E Leiserson. Systolic Arrays for (VLSI). In *Sparse Matrix Proceedings 1978*, volume 1, pages 256–282. Society for industrial and applied mathematics Philadelphia, PA, USA, 1979.

[101] Mitsuru Kusumoto, Takuya Inoue, Gentaro Watanabe, Takuya Akiba, and Masanori Koyama. A Graph Theoretic Framework of Recomputation Algorithms for Memory-Efficient Backpropagation. In *Advances in Neural Information Processing Systems (NeurIPS)*, pages 1161–1170. Curran Associates, Inc., 2019.

[102] Griffin Lacey, Graham W. Taylor, and Shawki Areibi. Stochastic Layer-Wise Precision in Deep Neural Networks. In Amir Globerson and Ricardo Silva, editors, *Proceedings of the Thirty-Fourth Conference on Uncertainty in Artificial Intelligence, UAI 2018, Monterey, California, USA, August 6-10, 2018*, pages 663–672. AUAI Press, 2018.

[103] Rasmus Munk Larsen and Tatiana Shpeisman. TensorFlow Graph Optimizations, 2019.

[104] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep Learning. *Nature*, 521(7553):436–444, 2015.

[105] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-Based Learning Applied to Document Recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[106] Dong-Hyun Lee et al. Pseudo-Label: The Simple and Efficient Semi-Supervised Learning Method for Deep Neural Networks. In *Workshop on challenges in representation learning, ICML*, 2013.

[107] Seunghoon Lee, Seunghyun Lee, and Byung Cheol Song. Improving Vision Transformers to Learn Small-Size Dataset From Scratch. *IEEE Access*, 10:123212–123224, 2022.

[108] Sil C. van de Leemput, Jonas Teuwen, Bram van Ginneken, and Rashindra Manniesing. MemCNN: A Python/PyTorch package for creating memory-efficient invertible neural networks. *Journal of Open Source Software*, 4(39):1576, 2019.

[109] Cheng Li. LLM-Analysis: Latency and Memory Analysis of Transformer Models for Training and Inference. https://github.com/cli99/llm-analysis, 2023.

[110] Mu Li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su. Scaling Distributed Machine Learning with the Parameter Server. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, page 583–598, USA, 2014. USENIX Association.

[111] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, and Soumith Chintala. PyTorch Distributed: Experiences on Accelerating Data Parallel Training. *Proceedings of the Very Large Data Base (VLDB) Endowment*, 13(12):3005–3018, Aug 2020.

[112] Xiuhong Li, Yun Liang, Shengen Yan, Liancheng Jia, and Yinghan Li. A Coordinated Tiling and Batching Framework for Efficient GEMM on GPUs. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming (PPoPP)*, page 229–241, New York, NY, USA, 2019. Association for Computing Machinery.

[113] Tailin Liang, John Glossner, Lei Wang, Shaobo Shi, and Xiaotong Zhang. Pruning and Quantization for Deep Neural Network Acceleration: A Survey. *Neurocomputing*, 461:370–403, 2021.

[114] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous Control with Deep Reinforcement Learning. *International Conference on Learning Representations (ICLR)*, 2016.

[115] Chenxi Liu, Barret Zoph, Maxim Neumann, Jonathon Shlens, Wei Hua, Li-Jia Li, Li Fei-Fei, Alan Yuille, Jonathan Huang, and Kevin Murphy. Progressive Neural Architecture Search. In *Proceedings of the European Conference on computer vision (ECCV)*, pages 19–34, 2018.

[116] Hanxiao Liu, Karen Simonyan, and Yiming Yang. DARTS: Differentiable Architecture Search. *International Conference on Learning Representations (ICLR)*, 2019.

[117] Liyuan Liu, Xiaodong Liu, Jianfeng Gao, Weizhu Chen, and Jiawei Han. Understanding the Difficulty of Training Transformers. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 5747–5763, Online, November 2020. Association for Computational Linguistics.

[118] Zhenhua Liu, Yunhe Wang, Kai Han, Wei Zhang, Siwei Ma, and Wen Gao. Post-Training Quantization for Vision Transformer. *Advances in Neural Information Processing Systems (NeurIPS)*, 34:28092–28103, 2021.

[119] Zhuang Liu, Mingjie Sun, Tinghui Zhou, Gao Huang, and Trevor Darrell. Rethinking the Value of Network Pruning. *International Conference on Learning Representations (ICLR)*, 2019.

[120] Ilya Loshchilov and Frank Hutter. SGDR: Stochastic Gradient Descent with Warm Restarts. *International Conference on Learning Representations (ICLR)*, 2017.

[121] Ilya Loshchilov and Frank Hutter. Decoupled Weight Decay Regularization. *International Conference on Learning Representations (ICLR)*, 2019.

[122] Renqian Luo, Fei Tian, Tao Qin, Enhong Chen, and Tie-Yan Liu. Neural Architecture Optimization. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems (NeurIPS)*, volume 31. Curran Associates, Inc., 2018.

[123] Sangkug Lym, Armand Behroozi, Wei Wen, Ge Li, Yongkee Kwon, and Mattan Erez. Mini-batch Serialization: CNN Training with Inter-layer Data Reuse. *Proceedings of Machine Learning and Systems (MLSys)*, 1:264–275, 2019.

[124] Kaifeng Lyu, Zhiyuan Li, and Sanjeev Arora. Understanding the Generalization Benefit of Normalization Layers: Sharpness Reduction. *Advances in Neural Information Processing Systems (NeurIPS)*, 2022.

[125] Matthew MacKay, Paul Vicol, Jimmy Ba, and Roger B Grosse. Reversible Recurrent Neural Networks. *Advances in Neural Information Processing Systems (NeurIPS)*, 31, 2018.

[126] Chris J Maddison, Andriy Mnih, and Yee Whye Teh. The Concrete Distribution: A Continuous Relaxation of Discrete Random Variables. *International Conference on Learning Representations (ICLR)*, 2017.

[127] Karttikeya Mangalam, Haoqi Fan, Yanghao Li, Chao-Yuan Wu, Bo Xiong, Christoph Feichtenhofer, and Jitendra Malik. Reversible Vision Transformers. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 10830–10840, 2022.

[128] Diana Marculescu, Dimitrios Stamoulis, and Ermao Cai. Hardware-Aware Machine Learning: Modeling and Optimization. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8. ACM, 2018.

[129] Stefano Markidis, Steven Wei Der Chien, Erwin Laure, Ivy Bo Peng, and Jeffrey S Vetter. NVIDIA Tensor Core Programmability, Performance & Precision. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 522–531. IEEE, 2018.

[130] Silvano Martello and Paolo Toth. *Knapsack Problems: Algorithms and Computer Implementations*. John Wiley & Sons, Inc., USA, 1990.

[131] Javier Mata, Ignacio de Miguel, Ramón J. Durán, Noemí Merayo, Sandeep Kumar Singh, Admela Jukan, and Mohit Chamania. Artificial intelligence (AI) methods in optical networks: A comprehensive survey. *Optical Switching and Networking*, 28:43–57, 2018.

[132] Peter Mattson, Christine Cheng, Cody Coleman, Greg Diamos, Paulius Micikevicius, David Patterson, Hanlin Tang, Gu-Yeon Wei, Peter Bailis, Victor Bittorf, David Brooks, Dehao Chen, Debojyoti Dutta, Udit Gupta, Kim Hazelwood, Andrew Hock,

Xinyuan Huang, Atsushi Ike, Bill Jia, Daniel Kang, David Kanter, Naveen Kumar, Jeffery Liao, Guokai Ma, Deepak Narayanan, Tayo Oguntebi, Gennady Pekhimenko, Lillian Pentecost, Vijay Janapa Reddi, Taylor Robie, Tom St. John, Tsuguchika Tabaru, Carole-Jean Wu, Lingjie Xu, Masafumi Yamazaki, Cliff Young, and Matei Zaharia. MLPerf Training Benchmark, 2020.

[133] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory F. Diamos, Erich Elsen, David García, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, and Hao Wu. Mixed Precision Training. In *International Conference on Learning Representations, (ICLR)*, 2018.

[134] Aaron Mok. ChatGPT could cost over $700,000 per day to operate. Microsoft is reportedly trying to make it cheaper. https://www.businessinsider.com/how-much-chatgpt-costs Accessed: 2023-05-01.

[135] Samuel G Müller and Frank Hutter. TrivialAugment: Tuning-free Yet State-of-the-Art Data Augmentation. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 774–782, 2021.

[136] Sharan Narang, Hyung Won Chung, Yi Tay, Liam Fedus, Thibault Fevry, Michael Matena, Karishma Malkan, Noah Fiedel, Noam Shazeer, Zhenzhong Lan, Yanqi Zhou, Wei Li, Nan Ding, Jake Marcus, Adam Roberts, and Colin Raffel. Do Transformer Modifications Transfer Across Implementations and Applications? In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 5758–5773, Online and Punta Cana, Dominican Republic, November 2021. Association for Computational Linguistics.

[137] James O' Neill. An Overview of Neural Network Compression. *arXiv preprint arXiv:2006.03669*, 2020.

[138] Yuval Netzer, Tao Wang, Adam Coates, Alessandro Bissacco, Bo Wu, and Andrew Y. Ng. Reading Digits in Natural Images with Unsupervised Feature Learning. In *NeurIPS Workshop on Deep Learning and Unsupervised Feature Learning 2011*, 2011.

[139] Timothy Nguyen, Zhourung Chen, and Jaehoon Lee. Dataset Meta-Learning from Kernel Ridge-Regression. *International Conference on Learning Representations (ICLR)*, 2021.

[140] Wei Niu, Jiexiong Guan, Yanzhi Wang, Gagan Agrawal, and Bin Ren. DNNFusion: Accelerating Deep Neural Networks Execution with Advanced Operator Fusion. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*, pages 883–898, 2021.

[141] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. BLEU: A Method for Automatic Evaluation of Machine Translation. In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics (ACL)*, page 311–318, USA, 2002. Association for Computational Linguistics.

[142] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In H. Wallach,

H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems (NeurIPS)*, pages 8024–8035. Curran Associates, Inc., 2019.

[143] Tim Pearce, Alexandra Brintrup, and Jun Zhu. Understanding Softmax Confidence and Uncertainty. *arXiv preprint arXiv:2106.04972*, 2021.

[144] Juan-Manuel Pérez-Rúa, Valentin Vielzeuf, Stéphane Pateux, Moez Baccouche, and Frédéric Jurie. MFAS: Multimodal Fusion Architecture Search. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 6966–6975, 2019.

[145] Hieu Pham, Melody Guan, Barret Zoph, Quoc Le, and Jeff Dean. Efficient Neural Architecture Search via Parameters Sharing. In Jennifer Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning (ICML)*, volume 80 of *Proceedings of Machine Learning Research*, pages 4095–4104. PMLR, 10–15 Jul 2018.

[146] Martin Popel and Ondřej Bojar. Training Tips for the Transformer Model. *Prague Bulletin of Mathematical Linguistics (PBML)*, 2018.

[147] Xipeng Qiu, Tianxiang Sun, Yige Xu, Yunfan Shao, Ning Dai, and Xuanjing Huang. Pre-trained Models for Natural Language Processing: A Survey. *Science China Technological Sciences*, 63(10):1872–1897, 2020.

[148] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language Models are Unsupervised Multitask Learners. *OpenAI blog*, 1(8):9,

2019.

[149] Jack W Rae, Sebastian Borgeaud, Trevor Cai, Katie Millican, Jordan Hoffmann, Francis Song, John Aslanides, Sarah Henderson, Roman Ring, Susannah Young, et al. Scaling Language Models: Methods, Analysis & Insights from Training Gopher. *arXiv preprint arXiv:2112.11446*, 2021.

[150] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. *The Journal of Machine Learning Research (JMLR)*, 21(1):5485–5551, 2020.

[151] Aswin Raghavan, Mohamed Amer, Sek Chai, and Graham Taylor. BitNet: Bit-Regularized Deep Neural Networks. *arXiv preprint arXiv:1708.04788*, 2017.

[152] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks. In *European Conference on Computer Vision (ECCV)*, pages 525–542. Springer, 2016.

[153] Sylvestre-Alvise Rebuffi, Alexander Kolesnikov, Georg Sperl, and Christoph H Lampert. iCaRL: Incremental Classifier and Representation Learning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2001–2010, 2017.

[154] Sashank J. Reddi, Satyen Kale, and Sanjiv Kumar. On the Convergence of Adam and Beyond. In *International Conference on Learning Representations (ICLR)*, 2018.

[155] Ao Ren, Tianyun Zhang, Shaokai Ye, Jiayu Li, Wenyao Xu, Xuehai Qian, Xue Lin, and Yanzhi Wang. ADMM-NN: An Algorithm-Hardware Co-Design Framework of DNNs Using Alternating Direction Method of Multipliers. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 925–938, 2019.

[156] Minsoo Rhu, Natalia Gimelshein, Jason Clemons, Arslan Zulfiqar, and Stephen W. Keckler. VDNN: Virtualized Deep Neural Networks for Scalable, Memory-Efficient Neural Network Design. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE Press, 2016.

[157] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Neurocomputing: Foundations of Research. *Nature*, pages 696–699, 1988.

[158] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015.

[159] Amit Sabne. XLA: Compiling Machine Learning for Peak Performance. 2020.

[160] Charbel Sakr, Yongjune Kim, and Naresh Shanbhag. Analytical Guarantees on Numerical Precision of Deep Neural Networks. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning (ICML)*, volume 70 of *Proceedings of Machine Learning Research*, pages 3007–3016. PMLR, 06–11 Aug 2017.

[161] Mark Sandler, Andrew G. Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. MobileNetV2: Inverted Residuals and Linear Bottlenecks. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 4510–4520, 2018.

[162] Shibani Santurkar, Dimitris Tsipras, Andrew Ilyas, and Aleksander Madry. How Does Batch Normalization Help Optimization? *Advances in Neural Information Processing Systems (NeurIPS)*, 31, 2018.

[163] Ozan Sener and Silvio Savarese. Active Learning for Convolutional Neural Networks: A Core-Set Approach. *International Conference on Learning Representations (ICLR)*, 2018.

[164] Hardik Sharma, Jongse Park, Naveen Suda, Liangzhen Lai, Benson Chau, Joon Kyung Kim, Vikas Chandra, and Hadi Esmaeilzadeh. Bit Fusion: Bit-Level Dynamically Composable Architecture for Accelerating Deep Neural Network. In *ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 764–775, 2018.

[165] Sam Shleifer and Eric Prokop. Using Small Proxy Datasets to Accelerate Hyperparameter Search. *arXiv preprint arXiv:1906.04887*, 2019.

[166] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

[167] Karen Simonyan and Andrew Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition. In *International Conference on Learning Representations (ICLR)*, 2015.

[168] Nimit Sharad Sohoni, Christopher Richard Aberger, Megan Leszczynski, Jian Zhang, and Christopher Ré. Low-Memory Neural Network Training: A Technical Report, 2019.

[169] Yisheng Song, Ting Wang, Puyu Cai, Subrota K Mondal, and Jyoti Prakash Sahoo. A Comprehensive Survey of Few-Shot Learning: Evolution, Applications, Challenges, and Opportunities. *ACM Computing Surveys*, Feb 2023. Just Accepted.

[170] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *Journal of Machine Learning Research (JMLR)*, 15(56):1929–1958, 2014.

[171] Dimitrios Stamoulis, Ting-Wu Rudy Chin, Anand Krishnan Prakash, Haocheng Fang, Sribhuvan Sajja, Mitchell Bognar, and Diana Marculescu. Designing Adaptive Neural Networks for Energy-Constrained Image Classification. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, page 1–8. IEEE Press, 2018.

[172] Felipe Petroski Such, Aditya Rawal, Joel Lehman, Kenneth Stanley, and Jeffrey Clune. Generative Teaching Networks: Accelerating Neural Architecture Search by Learning to Generate Synthetic Training Data. In *International Conference on Machine Learning (ICML)*, pages 9206–9216. PMLR, 2020.

[173] Ilia Sucholutsky and Matthias Schonlau. Soft-Label Dataset Distillation and Text Dataset Distillation. In *International Joint Conference on Neural Networks (IJCNN)*, pages 1–8, 2021.

[174] V. Sze, Y. Chen, T. Yang, and J. S. Emer. Efficient Processing of Deep Neural Networks: A Tutorial and Survey. *Proceedings of the IEEE*, 105(12):2295–2329, Dec 2017.

[175] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the Inception Architecture for Computer Vision. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2818–2826, 2016.

[176] Rohan Taori, Ishaan Gulrajani, Tianyi Zhang, Yann Dubois, Xuechen Li, Carlos Guestrin, Percy Liang, and Tatsunori B. Hashimoto. Stanford Alpaca: An Instruction-following LLaMA model. https://github.com/tatsu-lab/stanford_alpaca, 2023.

[177] Yi Tay, Mostafa Dehghani, Dara Bahri, and Donald Metzler. Efficient Transformers: A Survey. *ACM Computing Surveys*, 55(6):1–28, 2022.

[178] Seiya Tokui, Ryosuke Okuta, Takuya Akiba, Yusuke Niitani, Toru Ogawa, Shunta Saito, Shuji Suzuki, Kota Uenishi, Brian Vogel, and Hiroyuki Yamazaki Vincent. Chainer: A Deep Learning Framework for Accelerating the Research Cycle. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 2002–2011. ACM, 2019.

[179] Hugo Touvron, Matthieu Cord, Matthijs Douze, Francisco Massa, Alexandre Sablayrolles, and Hervé Jégou. Training Data-Efficient Image Transformers & Distillation

through Attention. In *International Conference on Machine Learning (ICML)*, pages 10347–10357. PMLR, 2021.

[180] Hugo Touvron, Matthieu Cord, Alexandre Sablayrolles, Gabriel Synnaeve, and Hervé Jégou. Going Deeper with Image Transformers. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 32–42, 2021.

[181] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. LLaMA: Open and Efficient Foundation Language Models. *arXiv preprint arXiv:2302.13971*, 2023.

[182] Dmitry Ulyanov, Andrea Vedaldi, and Victor Lempitsky. Instance Normalization: The Missing Ingredient for Fast Stylization. *arXiv preprint arXiv:1607.08022*, 2016.

[183] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention Is All You Need. *Advances in Neural Information Processing Systems (NeurIPS)*, 30, 2017.

[184] T. Veniat and L. Denoyer. Learning Time/Memory-Efficient Deep Architectures with Budgeted Super Networks. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 3492–3500, Los Alamitos, CA, USA, Jun 2018. IEEE Computer Society.

[185] Kuan Wang, Zhijian Liu, Yujun Lin, Ji Lin, and Song Han. HAQ: Hardware-Aware Automated Quantization with Mixed Precision. In *Proceedings of the IEEE/CVF*

Conference on Computer Vision and Pattern Recognition (CVPR), pages 8612–8620, 2019.

[186] Tongzhou Wang, Jun-Yan Zhu, Antonio Torralba, and Alexei A Efros. Dataset Distillation. *arXiv preprint arXiv:1811.10959*, 2018.

[187] Xiaofei Wang, Yiwen Han, Victor CM Leung, Dusit Niyato, Xueqiang Yan, and Xu Chen. Convergence of Edge Computing and Deep Learning: A Comprehensive Survey. *IEEE Communications Surveys & Tutorials*, 22(2):869–904, 2020.

[188] Hongxin Wei, Renchunzi Xie, Hao Cheng, Lei Feng, Bo An, and Yixuan Li. Mitigating Neural Network Overconfidence with Logit Normalization. *International Conference on Machine Learning (ICML)*, 2022.

[189] Wei Wen, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. Learning Structured Sparsity in Deep Neural Networks. *Advances in Neural Information Processing Systems (NeurIPS)*, 29, 2016.

[190] Paul J. Werbos. Backpropagation Through Time: What It Does and How to Do It. *Proceedings of the IEEE*, 78(10):1550–1560, Oct 1990.

[191] Felix Wiewel and Bin Yang. Condensed Composite Memory Continual Learning. In *2021 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE, 2021.

[192] Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8(3-4):229–256, 1992.

[193] Bichen Wu, Xiaoliang Dai, Peizhao Zhang, Yanghan Wang, Fei Sun, Yiming Wu, Yuandong Tian, Peter Vajda, Yangqing Jia, and Kurt Keutzer. FBNet: Hardware-Aware Efficient ConvNet Design via Differentiable Neural Architecture Search. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 10734–10742, 2019.

[194] Bichen Wu, Yanghan Wang, Peizhao Zhang, Yuandong Tian, Peter Vajda, and Kurt Keutzer. Mixed Precision Quantization of ConvNets via Differentiable Neural Architecture Search. *arXiv preprint arXiv:1812.00090*, 2018.

[195] H. Wu, G. Diamos, S. Cadambi, and S. Yalamanchili. Kernel Weaver: Automatically Fusing Database Primitives for Efficient GPU Computation. In *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 107–118, Dec 2012.

[196] Xiaoxia Wu, Ethan Dyer, and Behnam Neyshabur. When Do Curricula Work? *International Conference on Learning Representations (ICLR)*, 2021.

[197] Yuxin Wu and Kaiming He. Group Normalization. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 3–19, 2018.

[198] Han Xiao, Kashif Rasul, and Roland Vollgraf. Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms. *arXiv preprint arXiv:1708.07747*, 2017.

[199] Saining Xie, Ross Girshick, Piotr Dollar, Zhuowen Tu, and Kaiming He. Aggregated Residual Transformations for Deep Neural Networks. *IEEE Conference on Computer*

Vision and Pattern Recognition (CVPR), Jul 2017.

[200] Sirui Xie, Hehui Zheng, Chunxiao Liu, and Liang Lin. SNAS: Stochastic Neural Architecture Search. *International Conference on Learning Representations (ICLR)*, 2019.

[201] Ruibin Xiong, Yunchang Yang, Di He, Kai Zheng, Shuxin Zheng, Chen Xing, Huishuai Zhang, Yanyan Lan, Liwei Wang, and Tieyan Liu. On Layer Normalization in the Transformer Architecture. In Hal Daumé III and Aarti Singh, editors, *Proceedings of the 37th International Conference on Machine Learning (ICML)*, volume 119 of *Proceedings of Machine Learning Research*, pages 10524–10533. PMLR, 13–18 Jul 2020.

[202] Jingjing Xu, Xu Sun, Zhiyuan Zhang, Guangxiang Zhao, and Junyang Lin. Understanding and Improving Layer Normalization. *Advances in Neural Information Processing Systems (NeurIPS)*, 32, 2019.

[203] Xiaowei Xu, Yukun Ding, Sharon Xiaobo Hu, Michael Niemier, Jason Cong, Yu Hu, and Yiyu Shi. Scaling for edge inference of deep neural networks. *Nature Electronics*, 1(4):216, 2018.

[204] Tien-Ju Yang, Yu-Hsin Chen, and Vivienne Sze. Designing Energy-Efficient Convolutional Neural Networks using Energy-Aware Pruning. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 6071–6079, Los Alamitos, CA, USA, Jul 2017. IEEE Computer Society.

[205] Mingsheng Ying. Quantum computation, quantum theory and AI. *Artificial Intelligence*, 174(2):162–176, 2010. Special Review Issue.

[206] Sangdoo Yun, Dongyoon Han, Seong Joon Oh, Sanghyuk Chun, Junsuk Choe, and Youngjoon Yoo. CutMix: Regularization Strategy to Train Strong Classifiers with Localizable Features. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 6023–6032, 2019.

[207] Matthew D. Zeiler. ADADELTA: An Adaptive Learning Rate Method. *CoRR*, abs/1212.5701, 2012.

[208] Daochen Zha, Zaid Pervaiz Bhat, Kwei-Herng Lai, Fan Yang, Zhimeng Jiang, Shaochen Zhong, and Xia Hu. Data-centric Artificial Intelligence: A Survey. *arXiv preprint arXiv:2303.10158*, 2023.

[209] Biao Zhang, Behrooz Ghorbani, Ankur Bapna, Yong Cheng, Xavier Garcia, Jonathan Shen, and Orhan Firat. Examining Scaling and Transfer of Language Model Architectures for Machine Translation. In *International Conference on Machine Learning (ICML)*, pages 26176–26192. PMLR, 2022.

[210] Biao Zhang and Rico Sennrich. Root mean square layer normalization. *Advances in Neural Information Processing Systems (NeurIPS)*, 32, 2019.

[211] Hongyi Zhang, Moustapha Cisse, Yann N Dauphin, and David Lopez-Paz. mixup: Beyond Empirical Risk Minimization. *International Conference on Learning Representations (ICLR)*, 2018.

[212] Junzhe Zhang, Sai Ho Yeung, Yao Shu, Bingsheng He, and Wei Wang. Efficient Memory Management for GPU-based Deep Learning Systems. *arXiv preprint arXiv:1903.06631*, 2019.

[213] Xiangyu Zhang, Xinyu Zhou, Mengxiao Lin, and Jian Sun. ShuffleNet: An Extremely Efficient Convolutional Neural Network for Mobile Devices. In *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR)*, pages 6848–6856, 2018.

[214] Yifan Zhang, Bryan Hooi, Lanqing Hong, and Jiashi Feng. Self-Supervised Aggregation of Diverse Experts for Test-Agnostic Long-Tailed Recognition. *Advances in Neural Information Processing Systems (NeurIPS)*, 35:34077–34090, 2022.

[215] Bo Zhao and Hakan Bilen. Dataset Condensation with Differentiable Siamese Augmentation. In *International Conference on Machine Learning (ICML)*, 2021.

[216] Bo Zhao, Konda Reddy Mopuri, and Hakan Bilen. Dataset Condensation with Gradient Matching. *International Conference on Learning Representations (ICLR)*, 2021.

[217] Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, et al. A survey of large language models. *arXiv preprint arXiv:2303.18223*, 2023.

[218] Zhun Zhong, Liang Zheng, Guoliang Kang, Shaozi Li, and Yi Yang. Random Erasing Data Augmentation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, 2020.

[219] Yanqi Zhou, Sudip Roy, Amirali Abdolrashidi, Daniel Wong, Peter Ma, Qiumin Xu, Hanxiao Liu, Phitchaya Phothilimtha, Shen Wang, Anna Goldie, et al. Transferable Graph Optimizers for ML Compilers. *Advances in Neural Information Processing Systems (NeurIPS)*, 33:13844–13855, 2020.

[220] Jun-Yan Zhu, Taesung Park, Phillip Isola, and Alexei A Efros. Unpaired Image-to-Image Translation Using Cycle-Consistent Adversarial Networks. In *IEEE International Conference on Computer Vision (ICCV)*, pages 2242–2251, Oct 2017.

[221] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V Le. Learning Transferable Architectures for Scalable Image Recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR)*, pages 8697–8710, 2018.

Vita

Zixuan Jiang received a B.E. in Electronic Information Engineering from Zhejiang University, Hangzhou, China, in June 2018. He started to pursue his Ph.D. degree in the Department of Electrical and Computer Engineering at the University of Texas at Austin in August 2018, supervised by Prof. David Z. Pan. He interned at Microsoft Cloud + AI, Google Research, Google Ads, and Google Cloud during his Ph.D. studies. He received the 2021 IEEE Transactions on Computer-Aided Design (TCAD) Donald O. Pederson Best Paper Award.

His work revolves around enhancing the efficiency of the machine learning software stack, encompassing various aspects from algorithms to compilation. Furthermore, he has gained valuable experience in the design of machine learning accelerators. His current research interests primarily focus on optimizing machine learning solutions, aiming to make them more efficient and accessible, particularly from the perspective of machine learning compilation.

Permanent address: zxjiangprc@gmail.com

This dissertation was typeset with LaTeX[†] by the author.

[†]LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.