OPTIMIZATION AND AUTOMATION FOR EFFICIENT NEURAL ARCHITECTURE DESIGN

BY

YUHONG LI

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois Urbana-Champaign, 2024

Urbana, Illinois

Doctoral Committee:

Professor Deming Chen, Chair
Professor Nam Sung Kim
Professor Yuxiong Wang
Professor Jinjun Xiong, University at Buffalo
Dr. Debadeepta Dey, Microsoft Research
Assistant Professor Cong (Callie) Hao, Georgia Institute of Technology

ABSTRACT

As AI progresses, deep neural networks (DNNs) have become increasingly complex and resource-intensive, challenging their deployment on both cloud and edge systems. This dissertation addresses the critical challenges in neural architecture design by exploring various efficient search and optimization methods. It aims to deepen our understanding of the factors influencing DNN quality and develop automated processes to create optimized architectures for various applications, especially in resource-constrained settings.

Structured into five chapters, this dissertation aims to address three central inquiries:

- **Efficient exploration within a defined design space**: How do we navigate a predefined design space to not only explore efficiently but also identify and forge superior architectural designs?
- **Innovation beyond conventional boundaries**: Machine-led design, guided by predefined algorithms and data, often lacks the creative novelty of researchers who utilize abstract thinking and diverse experiences. How can we not only break traditional confines but also refine and elevate innovative designs using automation?
- **Harmonizing hardware efficiency with quality**: How do we strike a balance between the efficiency demands of the target hardware and the quality requirements of the model?

Our journey starts with EDD, where we introduce a hardware-aware differentiable neural architecture search. This novel method employs a differentiable formulation that integrates both model and hardware variables into a single search space, allowing for the improvement of efficiency. The key innovations include enhanced predictions on massive classification datasets and better efficiency in energy and computation speed on the target hardware,

leading to significant advancements in the quality of neural architectures searched.

However, we notice that the latency of neural networks is easier to predict for well-defined search spaces compared to the quality of neural architectures. Also, search spaces are not always representable in a continuous form, such as varying downsampling rates across different convolutional layers in a CNN, which introduces discrete decision points. To address this challenge, we introduce GenNAS, a search algorithm that utilizes synthetic regression-based few-shot learning tasks to obtain estimated scores of networks that are highly correlated with their quality on large datasets.

Furthermore, we observe that although GenNAS effectively navigates discrete search spaces and predicts accurately, evaluating each network individually remains time-consuming. We propose Eproxy, a sophisticated proxy-based approach that leverages self-supervised learning and few-shot techniques. By creating a more challenging synthetic proxy task, Eproxy enables faster evaluation and drastically reduces computational expenses. Moreover, we introduce a task search space that allows Eproxy to adapt to unconventional search spaces across various tasks.

Although the proposed NAS algorithms can perform accurate predictions within enormous search spaces, these spaces usually consist of conventional components, such as standard convolutional layers, which limit the potential for innovative breakthroughs. Based on our observations that current models fail to efficiently handle long-range dependencies, we have designed a novel architectural component called Structured Global Convolution (SGConv). This component employs a multi-scale sub-kernel strategy, which significantly enhances the capture of long-range dependencies. We further explore the search space of the sparsity of SGConv and a mixed architecture with both SGConv and the attention as the components.

By understanding the benefit of collaboration between human innovation and machine automation, we use the paradigms to explore the design of large language models (LLMs) using automation. Noticing that LLMs are primarily memory-bound due to the sequential nature of autoregressive decoding—where each token generated requires transferring large model parameters from memory, resulting in inefficient use of computational resources—we introduce Medusa. Medusa optimizes decoding efficiency by employing an adaptor that integrates additional decoding heads capable of predicting mul-

tiple subsequent tokens simultaneously, significantly enhancing throughput. We develop an automated framework that delivers the adapter from training to deployment. We further explore the search space such as the sparsity of the tree and the number of decoding heads. Moreover, we propose a simple hardware-aware profiling strategy for Medusa to predict its performance on the target devices with the consideration of batch sizes, sequence lengths, and sizes of LLMs.

In conclusion, this dissertation addresses neural architecture design challenges through innovative optimization and automation methods, advancing AI. By exploring differentiable searches, synthetic evaluations, and innovations like SGConv and the Medusa framework, it enriches our understanding of neural network efficiency and sets innovative standards for AI applications. Each chapter contributes to deep learning design, enhancing applications from the mobile to the cloud and promoting sustainable AI evolution.

ACKNOWLEDGMENTS

My Ph.D. journey has been an unforgettable and priceless experience. I am deeply grateful for the immense support and assistance I received from my advisors, friends, colleagues, and most importantly, my family throughout this journey.

First and foremost, I would like to express my heartfelt gratitude to my advisor, Prof. Deming Chen, for his trust and for inviting me to join the ES-CAD research group. His creativity, dedication, and passion for research have greatly influenced me and molded me into a passionate researcher. Without his guidance and support, my achievements, including this dissertation, would not have been possible. I would also like to extend my thanks to Prof. Jinjun Xiong, Prof. Cong 'Callie' Hao, and Dr. Debadeepta Dey, my mentors and collaborators during my PhD career, for their unwavering support, guidance, and encouragement. They are exceptional researchers with vast knowledge, innovative ideas, and tireless enthusiasm. I am honored to have Prof. Nam Sung Kim and Prof. Yuxiong Wang on my thesis committee. Their valuable feedback and suggestions have greatly contributed to the improvement of my dissertation.

Furthermore, I would like to express my gratitude to all my labmates in the ES-CAD research group for sharing their brilliant ideas and engaging in thought-provoking discussions. Special thanks to Prof. Wen-mei Hwu, Dr. Xiaofan Zhang, Dr. Xinheng Liu, and Dr. Yao Chen for their close collaboration in building the hardware-aware neural architecture search in Chapter 2. I am grateful to Prof. Pan Li for his contributions toward the regression-based neural architecture search idea in Chapter 3. I also appreciate Jiajie Li's insightful discussions and contributions to the efficient proxy neural architecture search in Chapter 4. Lastly, I would like to thank Tianle Cai, Zhengyang Geng, Hongwu Peng, Prof. Jason D. Lee, and Prof. Tri Dao for their efforts in building the parallel decoding accelerator in Chapter 6.

Finally, I would like to express my deepest gratitude to my parents, my fiancé, and her parents for their unconditional love and support. Words fall short of expressing my appreciation, and I know that they are always proud of me regardless of my achievements.

TABLE OF CONTENTS

CHAPTER 1 INTRODUCTION 1
1.1 Motivation 2
1.2 Contribution 4

CHAPTER 2 EFFICIENT DIFFERENTIABLE DNN: CO-SEARCHING ARCHITECTURE AND IMPLEMENTATION FOR EMBEDDED AI SOLUTIONS 11
2.1 Introduction 12
2.2 Related Works 13
2.3 EDD problem formulation 14
2.4 Device-specific formulation 21
2.5 Overall algorithm 23
2.6 Experiments 24
2.7 Summary 27

CHAPTER 3 REGRESSION-BASED GENERIC NEURAL ARCHITECTURE SEARCH 28
3.1 Introduction 29
3.2 Related Work 32
3.3 Proposed GenNAS 33
3.4 Experiments 38
3.5 Summary 48

CHAPTER 4 EXTENSIBLE AND EFFICIENT PROXY FOR NEURAL ARCHITECTURE SEARCH 49
4.1 Introduction 50
4.2 Our Approaches 54
4.3 Experiments 58
4.4 Summary 70

CHAPTER 5 COMBINING HUMAN INSIGHT AND AUTOMATION VIA STRUCTURED GLOBAL CONVOLUTION 71
5.1 Introduction 72
5.2 Related work 75
5.3 Design of Global Convolutional Models 76

5.4 Experiments 80
5.5 Summary 92

CHAPTER 6 AUTOMATED DESIGN OF DECODING ADAPTER TOWARDS LARGE LANGUAGE MODELS 93
6.1 Introduction 94
6.2 Related Work 96
6.3 Methodology 100
6.4 Experiments 109
6.5 Summary 132

CHAPTER 7 CONCLUSION 133
7.1 Summary of Contributions 133
7.2 Implications and Future Directions 134
7.3 Final Remarks 135

REFERENCES 136

CHAPTER 1

INTRODUCTION

The rapid growth of deep learning has led to the development of various neural architectures, which have demonstrated remarkable success in a wide range of applications, including computer vision [1, 2, 3, 4], natural language processing [5, 6, 7, 8], speech recognition [9, 10], and time-series forecasting [11, 12]. However, designing these architectures is often a laborious and time-consuming process, heavily reliant on the expertise and intuition of engineers and researchers. As deep learning applications keep growing in complexity and diversity, the need for design optimization and automation becomes increasingly critical. Furthermore, it is essential to explore new architecture designs that can efficiently adapt to the ever-evolving landscape of AI applications, particularly in resource-constrained environments and the emerging era of foundation models.

The field of neural architecture design has witnessed significant advancements in recent years, with the introduction of novel architectures such as Transformers [5], which have revolutionized natural language processing and have been adapted for various tasks, including computer vision [3] and time-series forecasting [11]. Other notable developments include the emergence of efficient Transformer variants, such as Linear Transformers [13], which aim to reduce the computational complexity while maintaining competitive quality. Moreover, task-specific architectures have been proposed to address the unique challenges of different domains, such as Graph Neural Networks (GNNs) [14, 15, 16] for graph data processing. Despite these advancements, the design of neural architectures remains a complex and resource-intensive process. Neural Architecture Search (NAS) [17, 18, 19] has emerged as a promising approach to automate the design process, potentially discovering innovative architectures that outperform human-designed counterparts and adapt to various tasks and performance constraints. However, NAS still faces several challenges, including the high computational cost of searching

for optimal architectures, the reliance on task-specific labels, and the need for effective proxies to evaluate candidate architectures. Furthermore, there is a growing need to explore and develop novel, out-of-the-box neural architectures that exhibit strong empirical performance across various tasks and provide new insights into the design principles and characteristics that contribute to their effectiveness. By addressing these challenges and expanding the horizons of neural architecture design, we can unlock the full potential of deep learning applications and push the boundaries of AI capabilities.

This dissertation aims to tackle these challenges by exploring automation and optimization techniques for neural architecture design, contributing to a deeper understanding of the factors that influence the quality of neural architectures. By investigating novel approaches to network search, developing efficient proxy tasks for NAS, discovering novel architectures based on interesting insights, and automatically optimizing the designed architectures, we enhance the practical applicability of neural architecture design and promote the development of innovative, high-quality neural networks for various applications.

Ultimately, the insights gained from this research will contribute to the growing knowledge in deep learning and advance the state-of-the-art in neural architecture design. By providing a deeper understanding of the factors that influence the performance of neural architectures and developing practical tools and techniques for automation and optimization, this dissertation aims to facilitate the development of innovative, high-performance neural networks that can adapt to the diverse challenges of real-world applications.

1.1 Motivation

1.1.1 Scaled-Up Model Size

One of the most prominent trends in AI is the increasing size of neural networks, which has led to significant improvements in quality across various tasks. In natural language processing (NLP), the progression from BERT [7] (110M/340M parameters) to state-of-the-art models like Llama [20] (ranging from 7B to 70B) has showcased the benefits of scaling up model size. Similarly, in computer vision (CV), the evolution from AlexNet [1] (60M param-

eters and trained on 1.3M data) to foundation models like ViT [3] (ranging from 86M to 632M parameters and trained on >300M data) has demonstrated the power of larger architectures. However, the growth in model size has also brought forth challenges in terms of computational resources, memory requirements, and energy consumption. The exploration and development of these large-scale models have become increasingly time-consuming, often requiring months of training on high-performance computing clusters. This has necessitated the development of efficient neural architecture design techniques that can optimize the quality-to-cost ratio and accelerate the discovery of powerful architectures.

1.1.2 Challenges in Neural Architecture Search

Neural Architecture Search (NAS) has emerged as a promising approach to automate the design of neural networks, but it faces several difficulties. One major challenge is the ability to predict the quality of scaled-up architectures based on their smaller counterparts. The relationship between architecture size and performance is often complex and non-linear, making it challenging to identify optimal configurations without extensive experimentation. Additionally, the search process itself can be extremely time-consuming, requiring significant computational resources to explore the vast design space. For instance, the seminal work on NAS, NASNet [17] consumed more than 2000 GPU hours to discover a high-quality DNN, which is prohibitively expensive for many researchers and practitioners.

1.1.3 Architectures for Diverse Tasks and Modalities

As AI expands its reach to diverse tasks and modalities, there is a growing need for specialized architectures that can effectively handle the unique characteristics of each domain. Graph Neural Networks (GNNs) [14, 15, 16] have been developed to process graph-structured data, while architectures like the Linear Transformer [13, 21, 22] have been proposed to efficiently handle long sequences in NLP tasks. Hybrid architectures [23], such as those combining convolutional and recurrent layers [24], have shown promise in tasks involving both spatial and temporal dependencies. Moreover, techniques like mixture-

of-experts [25, 26] have been introduced to further enhance the adaptability and generalization capabilities of neural networks. This increasing diversity of architectures highlights the complexity of modern neural architecture design and the need for systematic approaches to navigate this landscape.

The challenges posed by the scaling up of model sizes, the difficulties in neural architecture search, and the need for architectures tailored to diverse tasks and modalities underscore the importance of developing efficient and effective neural architecture design methodologies. This dissertation aims to address these challenges by exploring techniques for hardware-aware optimization, efficient neural architecture search, and the design of innovative architectures that push the boundaries of performance and adaptability across various domains.

1.2 Contribution

The dissertation unfolds across five primary chapters, each delving into distinct perspectives of neural architecture designs guided by three central inquiries:

- How can we efficiently navigate the vast landscape of architectural possibilities to uncover designs that excel in performance? This question seeks to develop strategies for intelligently exploring the design space, leveraging techniques such as differentiable NAS and hardware-aware optimization to identify high-performing configurations while minimizing computational overhead. (Covered by Chapters 2, 3, 4)

- How can we leverage human ingenuity and insights to break free from conventional design paradigms and push the boundaries of creativity in neural architecture design? Moreover, how can we harness the power of automation to refine and elevate these innovative designs further? This inquiry motivates us to explore the interplay between human creativity and machine optimization. (Covered by Chapters 5, 6)

- How can we achieve an optimal balance between the efficiency requirements imposed by the target hardware and the performance demands of the model? While predicting and benchmarking the scaled latency

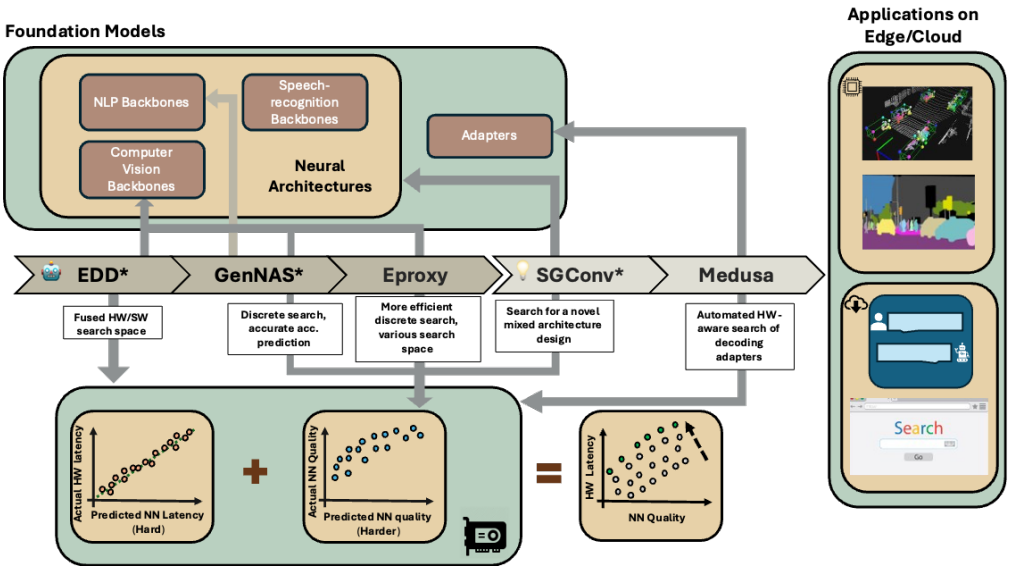


Figure 1.1: An overview of the dissertation, showcasing the exploration of efficient neural architecture designs and their applications across various domains. The figure illustrates the relation of each chapter. The proposed optimization and automation methods are applied to diverse domains, including computer vision, natural language processing, and speech recognition, demonstrating their versatility and potential for transforming AI model design and deployment on both edge and cloud platforms. The dissertation aims to address key challenges in neural architecture design, such as navigating the vast design space, pushing the boundaries of creativity, and balancing efficiency and performance, ultimately contributing to the advancement of AI capabilities across multiple fields.

of a model on specific hardware is relatively straightforward, estimating the accuracy of a neural architecture is a more complex task. This discrepancy arises due to the intricate interplay between the model's architecture, the dataset, and the training process, which makes it difficult to reliably predict the final accuracy without actually training and evaluating the model. (Chapters 2, 6 discuss the HW/SW co-design and Chapters 3, 4, 5 focus more on modeling accuracy of neural architectures.)

Through the development of practical tools, techniques, and insights, this dissertation seeks to deepen our understanding of the complex factors that shape the performance and efficiency of neural architectures, ultimately advancing the state-of-the-art in neural architecture design and paving the way for the creation of innovative, high-performance models that can readily adapt to the multifaceted challenges of real-world applications. The diagram shown in Figure 1.1 illustrates the applications of algorithms proposed in each chapter. The contributions for each main chapter are listed as follows:

- Chapter 2 introduces the Efficient Differentiable DNN (EDD) Search, which efficiently navigates the vast landscape of architectural possibilities by enabling simultaneous optimization of AI algorithms and hardware implementations. This novel approach fuses DNN search variables and hardware implementation variables into a single solution space, demonstrating a significant reduction in search time and increased adaptability to various hardware devices while minimizing computational overhead. By leveraging differentiable NAS and hardware-aware optimization techniques, EDD identifies high-performing configurations that excel in both accuracy and efficiency. In the experiments, EDD demonstrates its effectiveness by searching for three representative DNNs targeting low-latency GPU implementation and FPGA implementations with both recursive and pipelined architectures. Each model produced by EDD achieves similar accuracy as the best existing DNN models searched by state-of-the-art NAS methods on ImageNet but with superior accuracy obtained within a 12 GPU-hour search. The GPU-targeted DNN is 1.40× faster than the state-of-the-art Proxyless solution, and the FPGA-targeted DNN delivers 1.45× higher throughput than the state-of-the-art DNNBuilder solution, showcas-

ing EDD's ability to efficiently explore the design space and uncover high-performing architectures.

- In Chapter 3, we acknowledge the limitations of differentiable search methods when dealing with complex, non-differentiable search spaces. Moreover, predicting the neural architecture's accuracy is more challenging than benchmarking their hardware latency. To address this challenge, we introduce Generic Neural Architecture Search (GenNAS), a novel approach designed to efficiently explore and navigate these intricate search spaces that cannot be easily represented in a differentiable manner. GenNAS is a generic NAS framework that eliminates the reliance on task-specific labels for architecture evaluation. GenNAS adopts a unique approach by utilizing regression on a set of manually designed synthetic signal bases, which enables it to explore a more diverse range of search spaces that may not be easily represented in a differentiable manner. This transition from a differentiable to a discrete search strategy allows GenNAS to be more adaptable to various search spaces and tasks in both CV and NLP tasks. The synthetic signal bases serve as task-agnostic performance predictors, enabling GenNAS to swiftly identify promising architectures without the need for costly task-specific evaluations. Moreover, the discrete search strategy employed by GenNAS allows it to effectively navigate complex and non-differentiable search spaces, making it a versatile and efficient NAS framework. Through extensive experiments across a wide range of search spaces and tasks, GenNAS consistently achieves state-of-the-art performance while significantly reducing the computational overhead associated with traditional NAS methods.

- Although GenNAS is accurate in prediction, running it across vast search space is still time-consuming. Chapter 4 introduces Eproxy, a zero-cost (a neural architecture can be evaluated within seconds) proxy-based approach that significantly enhances the efficiency of neural architecture search by leveraging self-supervised learning and few-shot techniques. Proxy-based methods aim to reflect the model's true accuracy by utilizing a task requiring significantly less computation. Eproxy addresses the limitations of existing efficient proxies, which often struggle to adapt to various search spaces and tasks, by incorporat-

ing a barrier layer with randomly initialized frozen convolution parameters. This innovative feature adds non-linearities to the optimization spaces, enabling Eproxy to effectively discriminate architecture performance. To further optimize Eproxy's training settings, the chapter proposes the Discrete Proxy Search (DPS) method, which efficiently calibrates the predicted accuracy of Eproxy on a vast search space of over 5×10^{15} configurations using only a handful of benchmarked architectures on the target tasks. Eproxy can search over search spaces in the wild and its task-agnostic nature across various computer vision tasks. On the NAS-Bench-201 search space that consists of multiple tasks, Eproxy with DPS achieves comparable performance to the early stopping method while being 146 times faster in search cost. Moreover, Eproxy with DPS delivers strong results on the DARTS search space, achieving a Spearman ρ of 0.85 when transferred from CIFAR-10 to ImageNet, showcasing its transferability and task-agnostic nature.

- NAS excels at exploring vast search spaces to identify optimal architectures. However, most of the search spaces comprise pre-existing components such as convolutional and pooling layers, limiting NAS's ability to discover truly novel components. A crucial challenge lies in effectively integrating the innovative architectures we design with the power of search algorithms. In Chapter 5, we first identify two critical principles that contribute to the effective initialization of long convolutional models - efficient parameterization and decaying structure - we propose a simple yet powerful architectural design that breaks free from the confines of traditional local convolutions. This innovative approach showcases how human insights and observations can lead to novel architectural designs that excel in capturing long-range dependencies. We explore the component termed Structured Global Convolution (SGConv) across various domains, including computer vision, natural language processing, and speech recognition. Furthermore, we demonstrate how we can exploit the sparsity of the SGConv kernel to perform the automatic pruning after training. We also demonstrate that SGConv can be integrated with attention mechanisms to form a novel mixed architecture by searching. SGConv exemplifies the power of harnessing human creativity and machine optimization to push the

boundaries of architectural innovation, encouraging a bold step beyond conventional design paradigms. By combining our unique insights with the efficiency of automated search methods, we can refine and elevate these innovative designs, unlocking new possibilities in neural architecture search and optimization.

- We recognize the immense potential of leveraging automation to assist in developing new designs driven by human observations, particularly in the challenging era of large language models (LLMs). Chapter 6 addresses the critical need to design architectures that deliver high performance while operating within the constraints of various hardware platforms, aligning with the inquiry of balancing efficiency requirements and performance demands. We introduce Medusa, a generic adaptor for LLMs that tackles the memory-bound nature of LLMs through an automated framework for training and deploying adaptors in the decoding processes. By augmenting LLM inference with extra decoding heads to predict multiple subsequent tokens in parallel and utilizing a tree-based attention mechanism, Medusa constructs and verifies multiple candidate continuations simultaneously in each decoding step. This innovative approach not only breaks free from the conventional single-token prediction paradigm but also leverages automation to refine and elevate architectural design. Medusa's effectiveness is demonstrated through extensive experiments on models of various sizes and training procedures, including Vicuna-7B, 13B, 33B, and Zephyr-7B. The results show that Medusa can achieve a speedup of 2.3 to 2.8 times across different prompt types without compromising the quality of generation. Moreover, Medusa proposes several optimization strategies that enhance its performance and adaptability, such as the self-distillation approach, which enables Medusa to handle situations where no public data is available. By exploring the search space of the sparsity of the tree attention and the number of decoding heads, Medusa strikes a perfect balance between computational efficiency and generation quality. We also introduce a hardware study of Medusa and propose an analytical model to predict the performance of Medusa on target devices with varying batch sizes, sequence lengths, and model sizes. The hardware study underscores Medusa's impact by showing how it improves com-

pute resource utilization, enhancing both model runtime and hardware efficiency. By harnessing unique insights and leveraging automation, Medusa represents a pioneering approach to tackle the memory-bound nature of LLMs and paves the way for efficient and adaptable LLM inference across diverse hardware platforms.

CHAPTER 2

EFFICIENT DIFFERENTIABLE DNN: CO-SEARCHING ARCHITECTURE AND IMPLEMENTATION FOR EMBEDDED AI SOLUTIONS

High-quality AI solutions necessitate the efficient joint optimization of AI algorithms and their hardware implementations. However, navigating the vast landscape of architectural possibilities while considering hardware constraints poses significant challenges. In this chapter, we present the first fully simultaneous, **E**fficient **D**ifferentiable **D**NN architecture and implementation co-search (**EDD**) methodology. We formulate the co-search problem by integrating DNN search variables and hardware implementation variables and minimize both accuracy loss and hardware performance loss. The formulation is differentiable considering DNN and implementation variables and applies to various devices with different objectives. We demonstrate the effectiveness of our EDD methodology by searching for three representative DNNs, targeting low-latency GPU implementation and FPGA implementations with both recursive and pipelined architectures. Each model produced by EDD achieves similar accuracy to the best existing DNN models on ImageNet but with superior performance obtained within 12 GPU-hour searches. Our DNN targeting GPU is 1.40× faster than the state-of-the-art solution reported in ProxylessNAS [27], and our DNN targeting FPGA delivers 1.45× higher throughput than the state-of-the-art solution reported in DNNBuilder [28]. EDD approaches efficient and automatic search for hardware-friendly DNN design and implementation simultaneously, with an elegant and differentiable mathematical formulation. This chapter provides a comprehensive understanding of the EDD methodology and its implications for the future of efficient and adaptable AI solutions.

2.1 Introduction

AI algorithms have gained ever-increasing research interests and enabled superhuman solution quality in many domains. Remarkable achievements have been demonstrated for machine learning (ML) algorithm development, especially for deep neural networks (DNNs). A recent approach, neural architecture search (NAS) [29, 30, 31] has been widely successful in automatically developing DNNs that outperform human-crafted designs. On the other hand, the optimization techniques for high-performance implementations of AI algorithms on hardware are also being intensively studied. Such implementation techniques include kernel and DNN optimizations on GPUs and TPUs and accelerator designs on customizable hardware such as FPGAs and AI chips [28, 32, 33]. On top of the achievements, to further improve AI solution quality, AI algorithm designers and hardware developers begin to explore joint optimization opportunities. For example, hardware-aware NAS has been attracting more attention [27, 34, 35, 36, 37, 38]. Meanwhile, FPGA-oriented hardware/software co-design approaches [39, 33] focus on FPGA implementation characteristics and studied their influences on DNN design.

Despite many of the achievements of hardware-aware NAS and hardware/software co-design, there is still a large optimization opportunity missing: *the hardware implementation must be simultaneously searched during NAS*. For general-purpose computing devices such as GPUs and TPUs, implementation search means optimizing DNN implementations, for example, kernel fusion and memory access optimization. For reconfigurable devices such as FPGAs, implementation search means optimizing a customized DNN accelerator. Hardware implementation search not only provides more accurate performance data rather than estimated values, but more importantly, it provides instant guidance to DNN design during NAS. Currently, all existing works are missing the large design space of *implementation search* in their design flows, using estimated hardware performance from a fixed implementation [27, 34, 35, 36, 37, 38]. An initial discussion of the potential of simultaneous neural architecture and implementation co-search is carried out in [40], called NAIS, but the proposal stayed at conceptual level and no detailed methodology was provided. Inspired by [40], in this work, we propose a fully simultaneous DNN and implementation co-search methodology.

We summarize our contributions as follows:

- This is the first work that proposes a mathematical formulation to solve the simultaneous DNN architecture and hardware implementation co-search problem. The co-search formulation covers the variables for DNN architecture and hardware implementation, forming a fused solution space; the objective is to minimize the DNN accuracy loss and implementation performance loss simultaneously, which is differentiable with respect to both DNN and implementation variables.

- The formulation is unified and comprehensive. It is applicable on various hardware platforms such as GPUs, FPGAs, and dedicated accelerators; it can target various performance objectives such as latency, throughput, or energy; it also formulates resource usage considering resource sharing.

- Based on the formulation, we create an efficient differentiable DNN and implementation co-search methodology (EDD). To the best of our knowledge, this is the first work that applies fully simultaneous co-search for both high accuracy DNN and high performance implementation.

- We demonstrate our EDD methodology targeting three architectures: GPU, recursive FPGA accelerator, and pipelined FPGA accelerator. Each model achieves similar accuracy as the best existing DNNs on ImageNet but superior performance: our GPU-targeted DNN is 1.40× faster than the state-of-the-art Proxyless solution [27], and our FPGA-targeted DNN delivers 1.45× higher throughput than the state-of-the-art DNNBuilder solution [28].

2.2 Related Works

While NAS has been largely successful in delivering high accuracy DNN models, hardware-aware NAS, which more focuses on DNN hardware efficiency, has become important [27, 34, 35, 36, 37, 38]. Some works incorporate hardware latency into the objective of NAS, such as [27, 35, 36, 37], some treat latency as a hard constraint such as [34]. For search algorithm, [27] and [36]

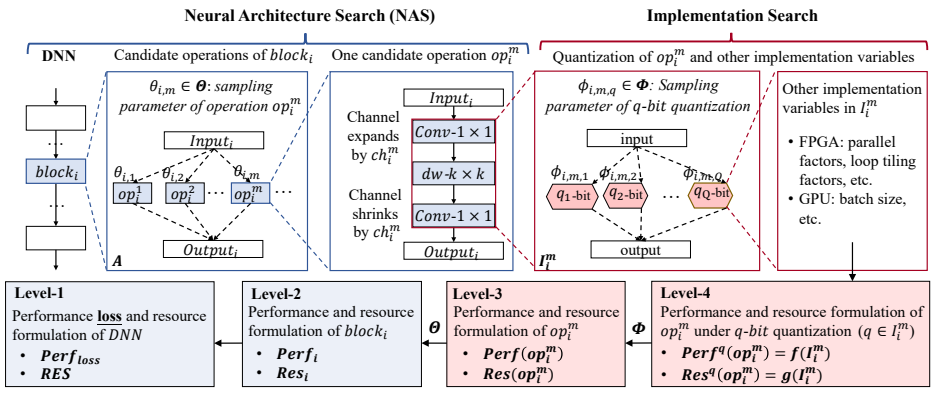


Figure 2.1: Our proposed differentiable DNN and implementation co-design (EDD) space.

use differentiable NAS approaches, where [27] uses binarized parameters to choose between different DNN branches, and [36] uses Gumbel-Softmax to convert discrete space to continuous space. The work [38] and [41] proposed mixed data precision for inference energy reduction.

On the other hand, various FPGA-based accelerators have been proposed for DNNs [42]. On top of that, FPGA-oriented hardware/software co-design approaches [39, 33] have been proposed, which focus on FPGA implementation characteristics and study their influences on DNN design. Specifically, [39] proposed a reinforcement learning based architecture search with FPGA implementation performance integrated into the reward function. Reference [33] proposed a bundle-based co-design methodology, where a bundle is the basic building block for both FPGA accelerator and DNN model. However, none of the previous works apply fully simultaneous DNN architecture and implementation co-search.

2.3 EDD problem formulation

The simultaneous DNN and implementation co-search problem fuses the design space of DNN architecture search and hardware implementation search, as shown in Figure 2.1. We collectively denote the variables used in DNN search and implementation search as A and I, respectively, and the fused space of co-search is $\{A, I\}$. The objective of DNN search is to minimize accuracy loss, denoted as Acc_{loss}. For implementation search, we define performance *loss*, denoted as $Perf_{loss}$, which users, such as end-to-end inference

latency, throughput, energy, DNN model complexity, etc can define. We denote the resource utilization during co-search as RES, and the resource upper-bound as RES_{ub}. The DNN and implementation co-search problem is to minimize accuracy loss Acc_{loss} and performance loss $Perf_{loss}$ simultaneously by searching $\{A, I\}$:

$$min : \mathcal{L} = Acc_{loss}(A, I) \cdot Perf_{loss}(I) + \beta \cdot C^{RES(I)-RES_{ub}}. \quad (2.1)$$

In Eq. 2.1, Acc_{loss} is a function of A and I; $Perf_{loss}$ and RES are functions of I. Resource upper-bound RES_{ub} is expressed in an exponent term to introduce large penalty when being violated. Worth noting, in the existing hardware-aware NAS approaches, only A is searched while I is *fixed* during NAS. In our proposed co-search formulation, I is *variable*, and A and I are *fused as one design space* $\{A, I\}$.

Among all NAS approaches searching for A, differentiable NAS, such as DARTS [18] and FBNet[36], has been proven to be GPU-hours efficient with appealing model accuracy. Motivated by differentiable NAS, in this work, we propose a unified differentiable formulation for both $\{A, I\}$: in Eq. 2.1, Acc_{loss} is differentiable with respect to A and I, and $Perf_{loss}$ and $RES(I)$ are differentiable with respect to I. In this way, by descending L on validation set as $\bigtriangledown_{\{A,I\}} L_{val}$, $\{A, I\}$ will be updated simultaneously.

Figure 2.1 shows our proposed overall differentiable design space. The blue blocks represent the DNN search space, while the red blocks represent the hardware implementation search space. The two spaces are merged into one as shown in Figure 2.2. We first introduce DNN search space in 2.3.1, and then introduce the merged design space with implementation in 2.3.2.

2.3.1 NAS Design Space

The differentiable NAS space is shown as the blue blocks in Figure 2.1. First, the DNN is composed of N basic building blocks, $block_i$, where $1 \leq i \leq N$. In this work, in order to design hardware-friendly DNNs and to reduce search time, we adopt the single-path DNN structure without branches [35].

Inside the *i-th* block, there are M candidate operations, denoted as op_i^m $(1 \leq m \leq M)$. We adopt the most commonly used DNN blocks in NAS approaches, called MBConv [34]. It is composed of sequential layers of *conv-*

1×1, *dwconv*-$k \times k$ and *conv*-1×1, where k is the kernel size. Between *conv*-1×1 and *dwconv*-$k \times k$, the number of channels expands/shrinks by a ratio of ch_i^m for operation op_i^m.

The output of each block is calculated based on the outputs of its M candidate operations; for example in [18], the output is the weighted sum of the M operations. In this work, we adopt the Gumbel-Softmax function in [36], where each operation op_m^i will be sampled from a sampling parameter $\theta_{i,m}$ following Gumbel-Softmax distribution, which converts the discrete non-differentiable sampling to continuous differentiable sampling. The sampling parameters $\theta_{i,m}$ organize a two-dimension $N \times M$ array, denoted as Θ, which is the primary DNN search variable $\Theta \in A$.

2.3.2 Implementation Formulation

As shown in the red block in Figure 2.1, each candidate operation op_i^m has its own implementation variables, forming an implementation search space I_i^m. The primary implementation variable is quantization q, i.e., data precision, since it has a large impact on DNN accuracy, implementation performance, and hardware resource. Rather than a train-and-quantize manner, the quantization shall be searched together with DNN structure to provide implementation performance feedback. Besides quantization, other implementation variables may be device oriented. For example, FPGA implementation design space includes parallelism, loop tiling factors, etc.

To formulate the final $Perf_{loss}$ and RES of the DNN in Eq. 2.1, we need to capture the intermediate performance and resource of each operation and DNN block using DNN and implementation variables. As shown in the bottom four blocks in Figure 2.1, there are four levels of formulations, each derived from their previous level:

- Level-4: we first formulate the quantization in a differentiable way; then, for each operation candidate op_i^m under q-bit, we formulate the performance as $Perf^q(op_i^m)$ and resource as $Res^q(op_i^m)$.
- Level-3: the performance and resource of op_i^m regardless of quantization, $Perf(op_i^m)$, and $Res(op_i^m)$, are derived from Level-4.
- Level-2: the performance and resource of *i-th* DNN block, $Perf_i$ and

Res_i, are derived from Level-3.

- Level-1: the final DNN performance *loss* and overall resource usage, $Perf_{loss}$ and RES, are derived from Level-2.

- Finally, $Perf_{loss}$ and RES will be plugged into Eq. 2.1 as the objective function during our EDD co-search.

In the following, we introduce the differentiable quantization and performance and resource formulations level by level.

2.3.2.1 Level-4: Differentiable Quantization

As shown in the red block in Figure 2.1, to enable differentiable quantization formulation, we create Q quantization paths for each operation op_i^m, indicating each operation has Q quantization choices, from q_1-*bit* to q_Q-*bit*. Each quantization scheme is associated with a sampling parameter from a Gumbel-Softmax distribution, denoted as $\phi_{i,m,q}$, generating a possibility for op_i^m to be quantized to *q-bit*. The $\phi_{i,m,q}$ organizes a three-dimension array of size $N \times M \times Q$, denoted as Φ. In this formulation, we have the flexibility to choose different quantizations for different layers of a DNN; such mixed precision computation can be well supported by reconfigurable hardware and dedicated accelerators.

Under *q-bit* quantization, the performance and resource of operation op_i^m, $Perf^q(op_i^m)$ and $Res^q(op_i^m)$, should be functions of implementation variables in I_i^m (including quantization q), expressed as $Perf^q(op_i^m) = f(I_i^m)$ and $Res^q(op_i^m) = g(I_i^m)$. Note that since one operation contains multiple layers as shown in Figure 2.1, for simplicity, we treat them as a whole: the *q-bit* quantization applies to all layers within op_i^m, and the latency and resource are the summation of all layers. Since $Perf^q(op_i^m)$ and $Res^q(op_i^m)$ largely vary with devices, we will discuss in Section 2.4.

Given differentiable DNN search variables Θ, differentiable quantization variables Φ and formulations under each quantization scheme, the DNN and implementation search space are fused as shown in Figure 2.2. The DNN and quantizations are searched through Θ and Φ; other implementation variables are searched through $Perf^q(op_i^m)$ and $Res^q(op_i^m)$, which are not related to accuracy.

2.3.2.2 From Level-4 to Level-3

Given array Φ, following Gumbel-Softmax sampling rule, the performance $Perf(op_i^m)$ and resource $Res(op_i^m)$ can be computed as the following:

$$Perf(op_i^m) = \sum_{1 \le q \le Q} GumbelSoftmax(\phi_{i,m,q} | \phi_{i,m}) \cdot Perf^q(op_i^m) \tag{2.2}$$

$$Res(op_i^m) = \sum_{1 \le q \le Q} GumbelSoftmax(\phi_{i,m,q} | \phi_{i,m}) \cdot Res^q(op_i^m) \tag{2.3}$$

where both $Perf(op_i^m)$ and $Res(op_i^m)$ are differentiable with respect to $\phi_{i,m,q}$. This is to compute the performance and resource expectation under different quantizations, which follows Gunbel-Softmax distribution with parameter $\phi_{i,m,q}$.

2.3.2.3 From Level-3 to Level-2

Similar to Eq. 2.2 and Eq. 2.3, given array Θ, the performance and resource of i-th DNN block, can be expressed as:

$$Perf_i = \sum_{1 \le m \le M} GumbelSoftmax(\theta_{i,m} | \theta_i) \cdot Perf(op_i^m) \tag{2.4}$$

$$Res_i = \sum_{1 \le m \le M} GumbelSoftmax(\theta_{i,m} | \theta_i) \cdot Res(op_i^m). \tag{2.5}$$

2.3.2.4 From Level-2 to Level-1 — Performance

Given the performance and resource of i-th DNN block, we can compute the overall DNN performance loss and resource usage, which needs to be tailored to search objectives and different devices.

First, if the overall objective is end-to-end latency, total energy or model complexity, the performance loss can be expressed using the summation of all DNN blocks as:

$$Perf_{loss} = \alpha \cdot \sum_{1 \le i \le N} Perf_i \tag{2.6}$$

where α scales $Perf_{loss}$ to the same magnitude of Acc_{loss} in Eq. 2.1.

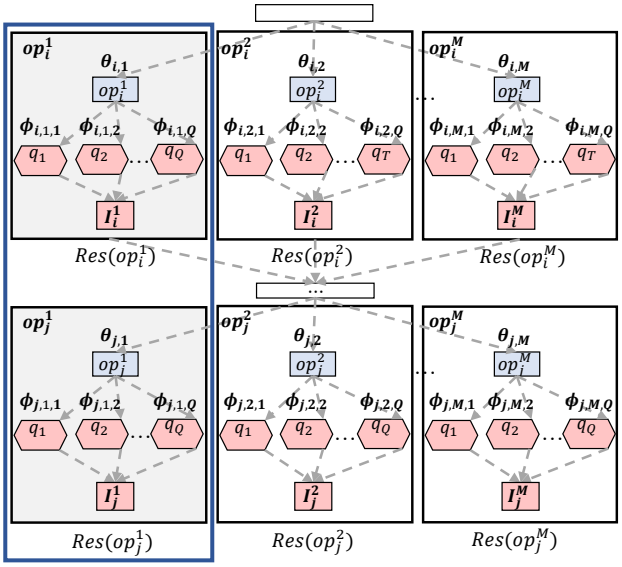


Figure 2.2: Fused design space including Θ (for DNN), Φ (for quantization) and other implementation variables in I.

If the objective is throughput, the performance loss can be the maximum latency of all blocks:

$$Perf_{loss} = \alpha \cdot max\{Perf_i\}, 1 \leq i \leq N. \tag{2.7}$$

Since getting the maximum value is a non-differentiable operation, we use a smooth maximum, Log-Sum-Exp (LSE) function [43], for differentiable approximation as:

$$Perf_{loss} = \alpha \cdot log \sum_{1 \leq i \leq N} e^{Perf_i}. \tag{2.8}$$

If there are multiple objectives, for example minimizing both latency and energy, as long as the objectives are not conflicting, we can simply let $Perf_{loss}$ be the production of different objectives.

2.3.2.5 From Level-2 to Level-1 — Resource

The formulation of overall resource usage has two situations: without and with resource sharing. Without sharing, the total resource can be computed

as the summation of all blocks:

$$RES = \sum_{1 \le i \le N} Res_i. \tag{2.9}$$

However, resource sharing is a very common scenario, especially in IP-based FPGA or ASIC accelerators. Figure 2.2 demonstrates a resource sharing scenario, where in this example, we assume the operation op_i^1 of $i\text{-}th$ block, and operation op_j^1 of $j\text{-}th$ block, will share a same piece of computing resource. For example, in FPGA or ASIC, it is a reusable IP. To allow sharing, the quantization and other implementation variables of op_i^1 and op_j^1 shall be the same[1]:

$$for\ \forall i, j \in [1, N],\ I_i^m = I_j^m = I^m. \tag{2.10}$$

Second, we discuss the resource estimation with resource sharing. As shown in Figure 2.3, $i\text{-}th$ row is the $i\text{-}th$ DNN block; the blue entries are the operations with the largest possibility to be selected. In this example, op_1^1 and op_i^1 are most likely to be chosen. Since they share the same computing resource $Res(op^1)$, it shall be counted only once. If the $m\text{-}th$ operation is not selected in any of the blocks, the resource $Res(op^m)$ shall not be counted.

To capture such scenario, we propose the following differentiable approximation for the resource usage for operation op^m, $Res(op^m)$, which is shared across blocks as:

$$Res(op^m) \leftarrow tanh(\sum_{1 \le i \le N} GumbelSoftmax(\theta_{i,m}|\theta_i) \cdot 1) \cdot Res(op^m). \tag{2.11}$$

In the above formula, $GumbelSoftmax(\theta_{i,m}|\theta_i) \cdot 1$ means the unit resource expectation of op_i^m in block i. To avoid operation resource being redundantly counted across block, we use $tanh(\sum)$ to suppress the maximum expectation of m-th operation to be 1 before multiplied by $Res(op^m)$.

Thus, the overall DNN resource is computed as:

$$RES = \sum_{1 \le m \le M} Res(op^m). \tag{2.12}$$

[1]Some accelerators allow different bit-width operations to share resource. In these cases constraint Eq. 2.10 is not needed.

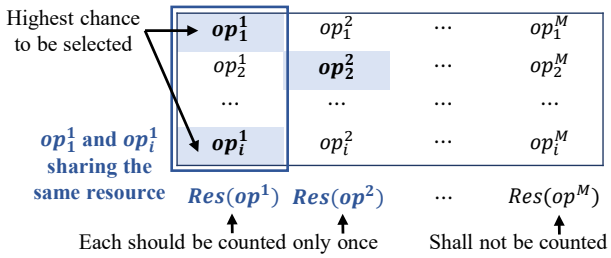


Figure 2.3: Demonstration of resource sharing.

2.4 Device-specific formulation

In this section, we discuss the device-specific formulations for the performance and resource of operation op_i^m under q-bit quantization, $Perf^q(op_i^m)$ and $Res^q(op_i^m)$.

2.4.1 FPGA

For FPGA implementation, we follow an IP-based accelerator architecture: for each operation op_i^m, there is a customizable IP instance to conduct its computation. There are typically two FPGA implementation architectures, recursive and pipelined:

- In the recursive architecture such as [32, 33], every DNN layer of the same type shares the same IP. The implementation objective is usually end-to-end latency. Thus, the $Perf_{loss}$ computation follows Eq. 2.6, and the RES follows either Eq. 2.11 or Eq. 2.12.
- In the pipelined architecture such as [28], every DNN layer has its own accelerators without resource sharing. The implementation target is usually throughput. The $Perf_{loss}$ computation follows Eq. 2.7, and the RES computation follows Eq. 2.9.

Either way, the operation performance $Perf^q(op_i^m)$ will be the operation latency. We let the operation resource $Res^q(op_i^m)$ to be the number of DSPs, which are usually the most critical resource on FPGA. To formulate $Perf^q(op_i^m)$ and $Res^q(op_i^m)$, we introduce additional implementation variables for FPGA, **the parallel factors** of the IPs, denoted as pf_i^m. Parallel factors describe the *parallelism* in FPGA, indicating how many multiplications can be done concurrently. In FPGA design, the parallelism usually

increases exponentially such as 64, 128, 256, etc. Therefore we use the exponential form of 2^{pf} to describe parallelism.

2.4.1.1 Latency — $Perf^q(op_i^m)$

As defined in Section 2.3.1, each operation op_i^m is composed of a set of sequential DNN layers such as convolution, batch normalization, and activation, and the latency and resource of op_i^m should be the summation of all layers. For an operation op_i^m with parallel factor of pf_i^m in q-bit, its latency can be approximated as:

$$Perf^q(op_i^m) = Lat^q(op_i^m) = \sum_{l \in op}(lat_l), \tag{2.13}$$

$$lat_l = \Phi(q) \times \begin{cases} 2^{-pf_i^m} \cdot k^2 \cdot h \cdot w \cdot c^{in} \cdot c^{out} & \text{if } l \text{ is conv.} \\ 2^{-pf_i^m} \cdot k^2 \cdot h \cdot w \cdot c^{in} & \text{if } l \text{ is dw-conv.} \\ 2^{-pf_i^m} \cdot h \cdot w \cdot c^{in} & \text{otherwise} \end{cases} \tag{2.14}$$

In the above equation, k is the convolution kernel size; h, w, c^{in} and c^{out} represent the data dimension of operation op_i^m; $\Phi(q)$ is the calibration for latency under bit-width of q. Intuitively, smaller bit-width leads to shorter latency because of less off-chip data movement and less computation. For simplification, we let $\Phi(q) = q$ to simulate such a phenomenon.

2.4.1.2 Resource — $Res^q(op_i^m)$

The resource (number of DSPs) of the IP with parallel factor of pf_i^m in q-bit can be approximated as:

$$Res^q(op_i^m) = \Psi(q) \times 2^{pf_i^m} \tag{2.15}$$

where $\Psi(q)$ is the calibration for resource under bit-width of q. On FPGA, the number of DSPs is non-linear to bit-width. For example, if the data precision is lower than 8-bit, then two multiplications can be calculated on one Xilinx DSP48, reducing the DSP usage by half; if the data precision is lower than 4-bit, we assume that multiplications are computed using Look-up-Tables (LUTs). Therefore, we use a piece-wise function to describe $\Psi(q)$:

$\Psi(q) = 1$ when $9 \leq q \leq 16$; $\Psi(q) = \frac{1}{2}$ when $5 \leq q \leq 8$; $\Psi(q) = 0$ when $q \leq 4$.

2.4.2 GPU

On GPUs, the most widely used performance metric is latency, so we let $Perf_{loss}$ to be the latency assuming the batch size is 1. We assume the resource is fixed given a GPU; in the future work, the resource shall also be captured since it affects the power. Since GPU latency is relatively easy to measure, we use normalized latency from directly measured values to represent inference latency under q-bit data precision. Therefore, $Perf^q(op_i^m)$ is a constant under a specific q. Currently, the GPU data precision is greatly restricted by the framework support. Since the current TensorRT only supports 8-bit fixed and 16-/32-bit floating data, we limit data precisions to be 8/16/32-bit for now but can easily extend to more data precisions. Meanwhile, since the current mixed precision inference has not been well supported by GPU development framework, we constrain the overall DNN to use the same data precision. Therefore, $\forall i, m$, we have $\phi_{i,m,q} = \phi_q$, which simplifies Eq. 2.2.

2.4.3 Dedicated Accelerators

Besides GPU and FPGA, there are also dedicated ASIC accelerators for efficient DNN implementation, such as Stripes [44], Loom [45], and Bit-Fusion [46], which are DNN accelerators that support dynamic data precisions efficiently. As an example, in the Loom [45] work, the computation latency and energy of convolution layers scale inversely and almost proportionally with the precisions of weights and activations. Our proposed method can be directly applied to such accelerators as well, by formulating the latency and energy of an operation op_m^i proportionally to data precision. We will leave this for future work.

2.5 Overall algorithm

First, the DNN and implementation variables are initialized, including the number of blocks N, the number of operation candidates M, and sampling

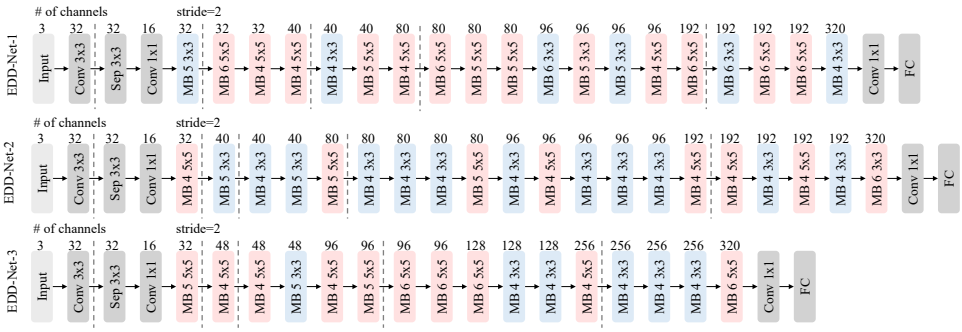


Figure 2.4: Architectures of three EDD-Net models. EDD-Net-1 targets GPU, EDD-Net-2 targets recursive FPGA accelerator, and EDD-Net-3 targets pipelined FPGA accelerator.

possibilities Θ and Φ. For recursive FPGA architecture, the algorithm initialize M IP instances, each with an initial parallel factor $pf_0 = log(\frac{RES_{ub}}{M})$. For pipelined FPGA architecture, the algorithm initializes $M \times N$ parallel factors for all the operation candidates, each $pf_0 = log(\frac{RES_{ub}}{M \times N})$. For different devices, only the initializations are different, and the remaining co-search follows the same procedure.

After initialization, the algorithm starts differentiable search for DNN and implementation following the similar bi-level approach in [18]; the major difference is in our back propagation, DNN architecture variables and implementation variables will be updated at the same time. It first fixes Θ, Φ and I, and updates DNN weights ω by minimizing the training loss on training dataset. Then, it fixes the DNN weights ω and updates Θ, Φ and I by descending Eq. 2.1 on the validation set. The algorithm repeats until DNN training converges or reaches a fixed number of epochs. Finally, the DNN needs to be trained from scratch on the target dataset, e.g. ImageNet, and the implementation variables, such as parallel factors, also need to be re-tuned for final implementation.

2.6 Experiments

We apply our EDD co-search on a subset of ImageNet dataset randomly sampled from 100 classes. The searched DNNs are trained from scratch on the entire ImageNet by replacing the final fully-connect layer from 100 to 1000 classes for collecting real ImageNet classification results. We run for a

Table 2.1: Comparisons with existing NAS solutions

	Test Error (%)		GPU Latency	FPGA Latency
	Top-1	Top-5	Titan RTX	ZCU102 [47]
Baseline Models				
GoogleNet	30.22	10.47	27.75 ms	13.25 ms
MobileNet-V2 [48]	28.1	9.7	17.87 ms	10.85 ms
ShuffleNet-V2 [49]	30.6	11.7	21.91 ms	NA
ResNet18	30.2	10.9	9.71 ms	10.15ms
Hardware-aware NAS Models				
MNasNet-A1 [34]	24.8	7.5	17.94 ms	8.78 ms
FBNet-C [36]	24.9	7.6	22.54 ms	12.21 ms
Proxyless-cpu [27]	24.7	7.6	21.34 ms	10.81 ms
Proxyless-Mobile [27]	25.4	7.8	21.23 ms	10.78 ms
Proxyless-gpu [27]	24.9	7.5	15.72 ms	10.79 ms
EDD-Net-1	25.3	7.7	**11.17 ms**	11.15 ms
EDD-Net-2	25.4	7.9	13.00 ms	**7.96 ms**

Table 2.2: EDD-Net-1 accuracy and latency on 1080 Ti

	32-bit Floating	**16-bit Floating**	**8-bit Integer**
Test Error	25.5%	25.3%	26.4%
Latency	2.83 ms	2.29 ms	1.74 ms

fixed 50 epochs during the EDD search. Initial DNN has 20 MBConv blocks, each with a skip-connection; each MBConv has a filter size of $\{3, 5, 7\}$ and an expansion ratio of $\{4, 5, 6\}$. During the search, for GPUs, the DNN weights are 8-/16-/32-bit and activations are 32-bit; for FPGAs, the DNN weights are 4-/8-/16-bit and activations are 16-bit fixed point.

We demonstrate our EDD methodology targeting three hardware architectures, each with a searched DNN model, called EDD-Net: (1) low-latency oriented GPU (EDD-Net-1); (2) recursive FPGA architecture (EDD-Net-2); (3) pipelined FPGA architecture (EDD-Net-3). The structures of the three DNNs are shown in Figure 2.4. Each model is produced through EDD within a 12-hour search on a P100 GPU.

First, for GPU-targeted EDD-Net-1, the algorithm suggests the 16-bit precision for weights for the combined objective function, including accuracy and

Table 2.3: Comparison of EDD-Net-3 with DNNBuilder [28]

	Top-1 Error (%)	**Top-5 Error (%)**	**Throughput (ZC706)**
VGG16	29.5	10.0	27.7 fps
EDD-Net-3	25.6	7.7	40.2 fps

latency. We compare EDD-Net-1 with the state-of-the-art hardware-aware NAS approaches, as shown in Table 2.1, where the GPU latency is tested on Titan RTX. First, it shows that, by targeting short GPU latency, EDD-Net-1 reaches a similar accuracy compared with the state-of-the-art DNN models while achieving the shortest inference latency, 11.17 ms, which is 1.4× faster than Proxyless-GPU [27], the previous best result reported through the NAS approach. Compared to other mobile-oriented NAS results as a reference, it is 2.0× faster than FBNet-C [36] and 1.6× faster than MNasNet [34]. Table 2.2 shows the accuracy and latency results of EDD-Net-1 on Nvidia 1080 Ti GPU after re-training and fine-tuning using TensorRT.

Second, we intend to compare FPGA-targeted EDD-Net-2 and EDD-Net-3 with existing FPGA/DNN co-design works such as [39] and [33], but neither of them provided accuracy results on ImageNet. Therefore, to demonstrate that our methodology applies well to FPGA and make a relatively fair comparison, for EDD-Net-2, which targets a recursive FPGA accelerator, we adopt the well-recognized CHaiDNN framework [47], which is also a recursive FPGA accelerator. The FPGA latency is collected by running various DNN models with CHaiDNN accelerators on ZCU102 FPGA as shown in Table 2.1, where ShuffleNet [49] is currently not supported by CHaiDNN. It shows that EDD-Net-2 delivers the shortest latency on FPGA among all the DNNs, 7.96 ms, which is 1.37× faster than ProxylessNet [27], 1.53× faster than FBNet [36] and 1.1× faster than MNasNet [34]. This result shows that our methodology generalizes well to FPGA and can search for FPGA-friendly DNNs effectively.

Third, EDD-Net-3 is searched targeting a pipelined FPGA accelerator. In this case, we limit the total number of DNN blocks because more DNN blocks require more resource and complicated memory control logic. In Fig 2.4, it shows that EDD-Net-3 is shallower but with more channels and larger kernels. We compare the throughput of EDD-Net-3 with a state-of-the-art

pipelined FPGA accelerator, DNNBuilder [28], on ZC706 FPGA with 900 DSPs. As shown in Table 2.3, under 16-bit fixed point, EDD-Net-3 achieves 1.45× higher throughput with a much higher accuracy.

2.7 Summary

In this chapter, we proposed a fully simultaneous, Efficient Differentiable DNN architecture and implementation co-search (EDD) methodology can target different hardware devices with different performance objectives. We formulated the co-search problem as an elegant differentiable mathematical formulation using DNN architecture search and hardware implementation variables, considering both accuracy and performance loss. In the experiments, we demonstrated three DNN models, targeting low-latency GPU, recursive FPGA accelerator and pipelined FPGA accelerator, respectively. Each model delivers the best performance with similar accuracy on ImageNet. The future works include GPU power and resource formulation and EDD search for dedicated accelerators.

CHAPTER 3

REGRESSION-BASED GENERIC NEURAL ARCHITECTURE SEARCH

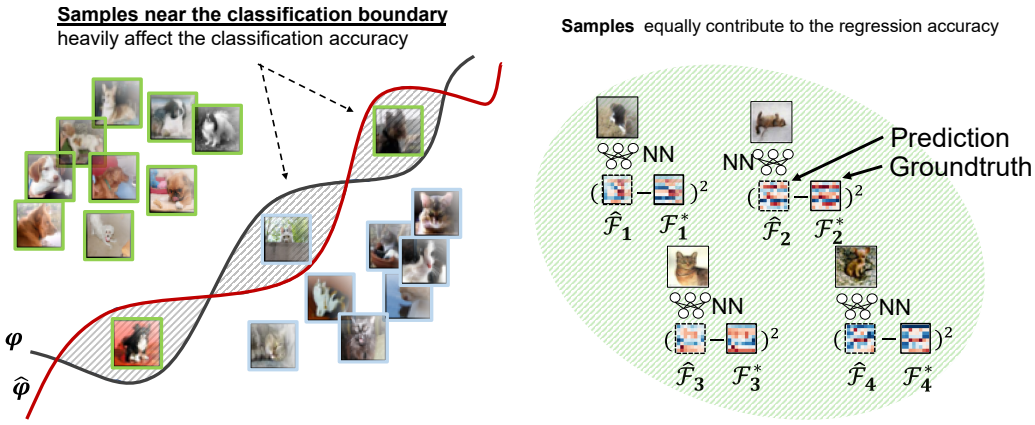
In the previous chapter, we introduced EDD, a novel method that utilizes differentiable NAS to co-search both hardware and software spaces simultaneously. However, we noticed that for more challenging search spaces, such as a CV backbone search space with different downsampling rates, the problem cannot be formulated as differentiable. Furthermore, while hardware latency can be easily obtained through benchmarking or simulation, predicting the accuracy of neural architectures remains a significant challenge. On the other hand, we observe that most existing neural architecture search (NAS) algorithms are dedicated to and evaluated by the downstream tasks, e.g., image classification in computer vision. However, extensive experiments have shown that prominent neural architectures, such as ResNet in computer vision and LSTM in natural language processing, are generally good at extracting patterns from the input data and perform well on different downstream tasks. This chapter attempts to answer two fundamental questions related to NAS. (1) Is it necessary to use the accuracy or quality of specific downstream tasks to evaluate and search for good neural architectures? (2) Can we perform NAS effectively and efficiently while agnostic to the downstream tasks? To answer these questions, we propose a novel and generic NAS framework, termed **Gen**eric **NAS** (GenNAS). GenNAS does not use task-specific labels but instead adopts *regression* on a set of manually designed synthetic signal bases for architecture evaluation. Such a self-supervised regression task can effectively evaluate the intrinsic power of an architecture to capture and transform the input signal patterns and allow sufficient usage of training samples. Extensive experiments across 13 CNN search spaces and one NLP space demonstrate the remarkable efficiency of GenNAS using regression, in terms of both evaluating the neural architectures (quantified by the ranking correlation Spearman's ρ between the approximated quality and the downstream task quality) and the convergence speed for training (within

a few seconds). For example, on NAS-Bench-101, GenNAS achieves 0.85 ρ while the existing efficient methods only achieve 0.38. We then propose an automatic task search to optimize the combination of synthetic signals using limited downstream-task-specific labels, further improving the quality of GenNAS. We also thoroughly evaluate GenNAS's generality and end-to-end NAS quality on all search spaces, outperforming almost all existing works with significant speedup. For example, on NASBench-201, GenNAS can find near-optimal architectures within 0.3 GPU hours.

3.1 Introduction

Most existing neural architecture search (NAS) approaches aim to find top-performing architectures on a specific downstream task, such as image classification [31, 17, 18, 27, 50], semantic segmentation [51, 52, 53], neural machine translation [54, 55, 56], or more complex tasks like hardware-software co-design [57, 33, 58, 59, 60]. They either directly search on the target task using the target dataset (e.g., classification on CIFAR-10 [17, 61]), or search on a *proxy* dataset and then transfer to the target one (e.g. CIFAR-10 to ImageNet) [29, 18]. However, extensive experiments show that prominent neural architectures are generally good at extracting patterns from the input data and perform well to different downstream tasks. For example, ResNet [62] being a prevailing architecture in computer vision, shows outstanding quality across various datasets and tasks [63, 64, 65] because of its advantageous architecture, the residual blocks. This observation motivates us to ask the first question: *Is there a generic way to search for and evaluate neural architectures without using the specific knowledge of downstream tasks?*

Meanwhile, we observe that most existing NAS approaches directly use the *final classification accuracy* as the metric for architecture evaluation and search, which has several major issues. First, the classification accuracy is dominated by the samples along the classification boundary, while other samples have clearer classification outcomes compared to the boundary ones (as illustrated in Figure 3.1a). Such phenomena can be observed in the limited number of effective support vectors in SVM [66], which also applies to neural networks because of the theory of neural tangent kernel [67]. Therefore, discriminating quality of classifiers needs many more samples than necessary



(a) Classification (b) Regression

Figure 3.1: For classification, only samples near the decision boundary determine the classification accuracy. For regression, all samples equally contribute to the regression accuracy. Therefore, regression is better at leveraging all training samples than classification to achieve faster convergence.

(the indeed effective ones), causing a big waste. Second, a classifier tends to discard valuable information, such as finer-grained features and spatial information, by transforming input representations into categorical labels. This observation motivates us to ask the second question: *Is there a more effective way to make sufficient use of input samples and better capture valuable information?*

To answer the two fundamental questions for NAS, in this work, we propose a **Gen**eric **N**eural **A**rchitecture **S**earch method, termed **GenNAS**. GenNAS adopts a **regression-based proxy task** using **downstream-task-agnostic synthetic signals** for network training and evaluation. It can efficiently (with near-zero training cost) and accurately *approximate* the neural architecture quality.

Insights. First, as opposed to classification, a regression can efficiently make full use of all the input samples, which equally contribute to the regression accuracy (Figure 3.1b). Second, regression on properly-designed synthetic signals is essentially evaluating the *intrinsic representation power* of neural architectures, which is to capture and distinguish fundamental data patterns that are agnostic to downstream tasks. Third, such representation power is heavily reflected in the *intermediate data* of a network (as we will show in the experiments), which are regrettably discarded by classification.

Approach. First, we propose a *regression proxy task* as the supervising task to train, evaluate, and search for neural architectures (Figure 3.2). Then, the searched architectures will be used for the target downstream tasks. We are the first to propose a self-supervised regression proxy task instead of classification for NAS. Second, we propose to use *unlabeled synthetic data* (e.g., sine and random signals) as the groundtruth (Figure 3.3) to measure neural architectures' intrinsic capability of capturing fundamental data patterns. Third, to further boost NAS quality, we propose a weakly-supervised automatic proxy task search with only a handful of groundtruth architecture quality (e.g. 20 architectures) to determine the best proxy task, i.e., the combination of synthetic signal bases, targeting a specific downstream task, search space, and/or dataset (Figure 3.4).

GenNAS Evaluation. The efficiency and effectiveness of NAS are dominated by *neural architecture evaluation*, which directs the search algorithm towards top-performing network architectures. To quantify how accurate the evaluation is, one widely used indicator is the network quality *Ranking Correlation* [68] between the prediction and groundtruth ranking, defined as Spearman's Rho (ρ) or Kendall's Tau (τ). The ideal ranking correlation is 1 when the approximated and groundtruth rankings are exactly the same; achieving large ρ or τ can significantly improve NAS quality [69, 70, 71]. Therefore, in the experiments (Sec. 3.4), we evaluate GenNAS using the ranking correlation factors it achieves and then show its end-to-end NAS quality in finding the best architectures. Extensive experiments are done on 13 CNN search spaces and one NLP space [72]. Trained by the regression proxy task using only a single batch of unlabeled data within a few seconds, GenNAS significantly outperforms all existing NAS approaches on almost all the search spaces and datasets. For example, GenNAS achieves 0.87 ρ on NASBench-101 [73], while Zero-Cost NAS [19], an efficient proxy NAS approach, only achieves 0.38. On end-to-end NAS, GenNAS generally outperforms others with a large speedup. This implies that the insights behind GenNAS are plausible and that our proposed regression-based task-agnostic approach is generalizable across tasks, search spaces, and datasets.

Contributions. We summarize our contributions as follows:

- To the best of our knowledge, GenNAS is the first NAS approach using regression as the self-supervised proxy task instead of classification for

neural architecture evaluation and search. It is agnostic to the specific downstream tasks and can significantly improve training and evaluation efficiency by fully utilizing only a handful of unlabeled data.

- GenNAS uses synthetic signal bases as the groundtruth to measure the intrinsic capability of networks that captures fundamental signal patterns. Using such unlabeled synthetic data in regression, GenNAS can find the generic task-agnostic top-performing networks and can apply to any new search spaces with zero effort.
- An automated proxy task search to further improve GenNAS quality.
- Thorough experiments show that GenNAS outperforms existing NAS approaches by large margins in terms of ranking correlation with near-zero training cost, across 13 CNN and one NLP space *without* proxy task search. GenNAS also achieves state-of-the-art quality for end-to-end NAS with orders of magnitude of speedup over conventional methods.
- With proxy task search being optional, GenNAS is fine-tuning-free, highly efficient, and can be easily implemented on a single customer-level GPU.

3.2 Related Work

NAS Evaluation. Network architecture evaluation is critical in guiding the search algorithms of NAS by identifying the top-performing architectures, which is also a challenging task with intensive research interests. Early NAS works evaluated the networks by training from scratch with tremendous computation and time cost [29, 31]. To expedite, weight-sharing among the subnets sampled from a supernet is widely adopted [18, 71, 27, 74, 75]. However, due to the poor correlation between the weight-sharing and the final quality ranking, weight-sharing NAS can easily fail even in simple search spaces [76, 77]. Yu et al. [78] further pointed out that without accurate evaluation, NAS runs in a near-random fashion. Recently, zero-cost NAS methods [79, 19, 80, 81] have been proposed, which score the networks using their initial parameters with only one forward and backward propagation. Despite

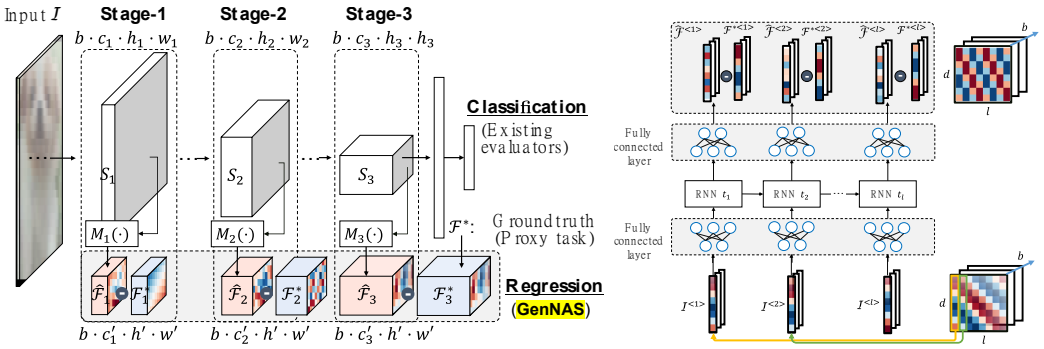
the significant speed up, they fail to identify top-performing architectures in large search spaces such as NASBench-101. To detach the time-consuming network evaluation from NAS, several benchmarks are developed with fully-trained neural networks within the NAS search spaces [82, 73, 77, 72, 83], so that researchers can assess the search algorithms alone in the playground.

NAS Transferability. To improve search efficiency, proxy tasks are widely used, on which the architectures are searched and then transferred to target datasets and tasks. For example, the CIFAR-10 classification dataset seems to be a good proxy for ImageNet [29, 18]. Kornblith et al. [84] studied the transferability of 16 classification networks on 12 image classification datasets. NASBench-201 [77] evaluated the ranking correlations across three popular datasets with 15625 architectures. Liu et al. [85] studied the architecture transferability across supervised and unsupervised tasks. Nevertheless, training on a downsized proxy dataset is still inefficient (e.g. a few epochs of full-blown training [85]). In contrast, GenNAS significantly improves the efficiency by using a single batch of data while maintaining extremely good generalizability across different search spaces and datasets.

Self-supervised Learning. Self-supervised learning is a form of unsupervised learning, that the neural architectures are trained with automatically generated labels to gain a good degree of comprehension or understanding [86, 87, 88, 89, 85]. Liu et al. [85] recently proposed three unlabeled classification proxy tasks, including rotation prediction, colorization, and solving jigsaw puzzles, for neural network evaluation. Though promising, this approach did not explain why such manually designed proxy tasks are beneficial and still used classification for training with the entire dataset. In contrast, GenNAS uses regression with only a single batch of synthetic data.

3.3 Proposed GenNAS

In Section 3.3.1, we introduce the main concepts of task-agnostic GenNAS: 1) the proposed regression proxy task for both CNN architectures and recurrent neural network (RNN) architectures; 2) the synthetic signal bases used for representing the fundamental data patterns as the proxy task. We introduce the automated proxy task search in Section 3.3.2.



(a) CNN regression (b) RNN regression

Figure 3.2: Regression architectures on CNNs and RNNs. (a) On CNNs, we remove the final classifier and extract multiple stages of the intermediate feature map for training. (b) On RNNs, we construct a many-to-many regression task where the input and output tensors have the same size.

3.3.1 GenNAS

Training using unlabeled regression is the key that GenNAS being agnostic to downstream tasks. Based on the insights discussed in Section 3.1, the *principle* of designing the regression architecture is to *fully utilize the abundant intermediate information* rather than the final classifier.

3.3.1.1 Regression on CNNs.

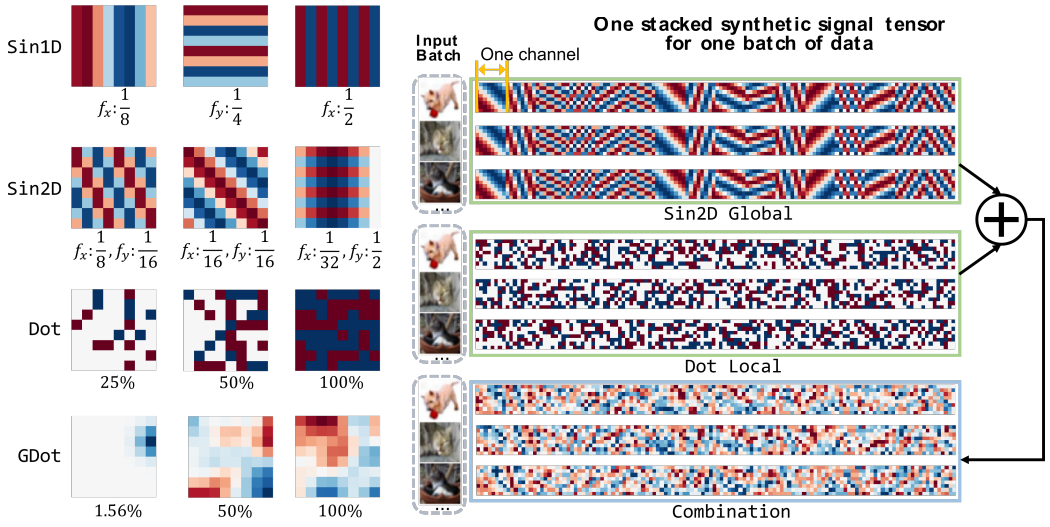
Empirical studies show that CNNs learn fine-grained high-frequency spatial details in the early layers and produce semantic features in the late layers [90]. Following this principle, as shown in Figure 3.2a, we construct a Fully Convolutional Network (FCN) [91] by removing the final classifier of a CNN, and then extract the FCN's intermediate feature maps from *multiple stages*. We denote the number of stages as N. **Inputs**. The inputs to the FCN are unlabeled real images, shaped as a tensor $\mathcal{I} \in \mathbb{R}^{b \times 3 \times h \times w}$, where b is the batch size, and h and w are the input image size. **Outputs**. From each stage i ($1 \leq i \leq N$) of the FCN, we first extract a feature map tensor, denoted by $\mathcal{F}_i \in \mathbb{R}^{b \times c_i \times h_i \times w_i}$, and reshape it as $\hat{\mathcal{F}}_i \in \mathbb{R}^{b \times c_i' \times h' \times w'}$ through a convolutional layer M_i by $\hat{\mathcal{F}}_i = M_i(\mathcal{F}_i)$ (with downsampling if $w_i > w'$ or $h_i > h'$). The outputs are the tensors $\hat{\mathcal{F}}_i$, which encapsulate the captured signal patterns from different stages. **Groundtruth**. We construct a synthetic signal tensor for each stage as the groundtruth, which serves as part

of the *proxy task*. A synthetic tensor is a combination of multiple synthetic signal bases, denoted by $\mathcal{F}_i^*$. We compare $\hat{\mathcal{F}}_i$ with $\mathcal{F}_i^*$ for training and evaluating the neural architectures. During training, we use MSE loss defined as $\mathcal{L} = \sum_{i=1}^{N} \mathbf{E}[(\mathcal{F}_i^* - \hat{\mathcal{F}}_i)^2]$; during validation, we adjust each stage's output importance as $\mathcal{L} = \sum_{i=1}^{N} \frac{1}{2^{N-i}} \mathbf{E}[(\mathcal{F}_i^* - \hat{\mathcal{F}}_i)^2]$ since the feature map tensors of later stages are more related to the downstream task's quality. The detailed configurations of N, h', w', and c_i' are provided in the experiments.

3.3.1.2 Regression on RNNs.

The proposed regression proxy task can be similarly applied to NLP tasks using RNNs. Most existing NLP models use a sequence of word-classifiers as the final outputs, whose evaluations are thus based on the word classification accuracy [92, 93, 5]. Following the same principle for CNNs, we design a many-to-many regression task for RNNs as shown in Figure 3.2b. Instead of using the final word-classifier's output, we extract the output tensor of the intermediate layer before it. **Inputs**. For a general RNN model, the input is a random tensor $\mathcal{I} \in \mathbb{R}^{l \times b \times d}$, where l is the sequence length, b is the batch size, and d is the length of input/output word vectors. Given a sequence of length l, the input to the RNN each time is one slice of the tensor $\mathcal{I}$, denoted by $\mathcal{I}^{(i)} \in \mathbb{R}^{b \times d}$, $1 \leq i \leq l$. **Outputs**. The output is $\hat{\mathcal{F}} \in \mathbb{R}^{l \times b \times d}$, where a slice of $\hat{\mathcal{F}}$ is $\hat{\mathcal{F}}^{(i)} \in \mathbb{R}^{b \times d}$. **Groundtruth**. Similar to the CNN case, we generate a synthetic signal tensor $\mathcal{F}^*$ as the proxy task groundtruth.

The proxy task for regression aims to capture the task-agnostic intrinsic learning capability of the neural architectures, i.e., representing various fundamental data patterns. For example, good CNNs must be able to learn different frequency signals to capture image features [94]. Here, we design four types of synthetic signal basis: (1) 1-D frequency basis (`Sin1D`); (2) 2-D frequency basis (`Sin2D`); (3) Spatial basis (`Dot` and `GDot`); (4) Resized input signal (`Resize`). `Sin1D` and `Sin2D` represent frequency information, `Dot` and `GDot` represent spatial information, and `Resize` reflects the CNN's scale-invariant capability. The combinations of these signal bases, especially with different sine frequencies, can represent a wide range of complicated real-world signals [95]. If a network architecture is good at learning such signal basis and their simple combinations, it is more likely to be able to capture real-world signals from different downstream tasks.



(a) Examples of signal bases (b) Examples of synthetic signal tensors

Figure 3.3: (a) Examples of synthetic signal bases (2D feature maps). (b) Examples of the synthetic signal tensors by stacking 2D feature maps along the channel dimension for CNN architectures.

Figure 3.3a depicts examples of synthetic signal bases, where each base is a 2D signal feature map. `Sin1D` is generated by $\sin(2\pi fx+\phi)$ or $\sin(2\pi fy+\phi)$, and `Sin2D` is generated by $\sin(2\pi f_x x+2\pi f_y y+\phi)$, where x and y are pixel indices. `Dot` is generated according to biased Rademacher distribution [96] by randomly setting $k\%$ pixels to ± 1 on zeroed feature maps. `GDot` is generated by applying a Gaussian filter with $\sigma = 1$ on `Dot` and normalizing between ± 1. The synthetic signal tensor $\mathcal{F}^*$ (the proxy task groundtruth) is constructed by stacking the 2D signal feature maps along the channel dimension (CNNs) or the batch dimension (for RNNs). Figure 3.3b shows examples of stacked synthetic tensor $\mathcal{F}^*$ for CNN architectures. Within one batch of input images, we consider two settings: `global` and `local`. The `global` setting means that the synthetic tensor is the same for all the inputs within the batch, as the `Sin2D Global` in Figure 3.3b, aiming to test the network's ability to capture invariant features from different inputs; the `local` setting uses different synthetic signal tensors for different inputs, as the `Dot Local` in Figure3.3b, aiming to test the network's ability to distinguish between images. For CNNs, the real images are only used by `resize`, and both `global` and `local` settings are used. For RNNs, we only use synthetic signals and the `local` setting because resizing natural language or time series, the typical input of RNNs, does not make as much sense as resizing images for CNNs.

3.3.2 Proxy Task Search

While the synthetic signals can express generic features, the importance of these features for different tasks, NAS search spaces, and datasets may differ. Therefore, we further propose a weakly-supervised proxy task search, to automatically find the best synthetic signal tensor, i.e., the best combination of synthetic signal bases. We define the *proxy task search space* as the parameters for generating synthetic signal tensors. As illustrated in Figure 3.4, first, we randomly sample a small subset (e.g., 20) of the neural architectures in the NAS search space and obtain their groundtruth ranking on the target task (e.g., image classification). We then train these networks using different proxy tasks and calculate the quality ranking correlation ρ of the proxy and the target task. We use the regularized tournament selection evolutionary algorithm [31] to search for the task that results in the largest ρ, where ρ is the fitness function.

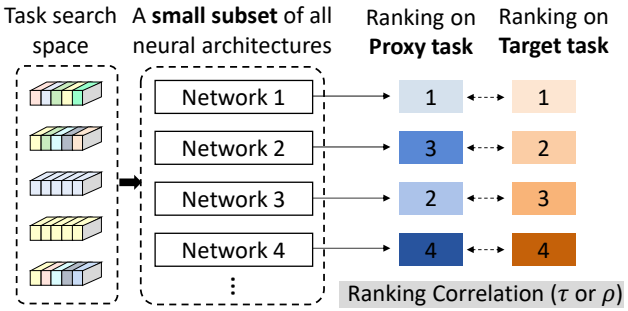


Figure 3.4: Proxy task search.

Proxy Task Search Space. We consider the following parameters as the proxy task search space. (1) Noise. We add noise to the input data following the distribution of parameterized Gaussian or uniform distribution. (2) The number of channels for each synthetic signal tensor (c_i in $\mathcal{F}_i^* \in \mathbb{R}^{b \times c_i \times h' \times w'}$) can be adjusted. (3) Signal parameters, such as the frequency f and phase ϕ in `Sin`, can be adjusted. (4) Feature combination. Each synthetic signal tensor uses either `local` or `global`, and tensors can be selected and summed up. Detailed parameters can be found in the supplemental material.

3.4 Experiments

We perform the following evaluations for GenNAS. First, to show the *true power of regression*, we use manually designed proxy tasks *without task search* and apply the same proxy task on all datasets and search spaces. We demonstrate that the *GenNAS generally excels in all different cases with zero task-specific cost*, thanks to unlabeled self-supervised regression proxy task. Specifically, in Section 3.4.1, we analyze the effectiveness of the synthetic signal bases and manually construct two sets of synthetic tensors as the baseline proxy tasks; in Section 3.4.2, we extensively evaluate the proposed regression approach in 13 CNN search spaces and one NLP search space. Second, in Section 3.4.3, we evaluate the proxy task search and demonstrate the remarkable generalizability by applying one searched task to all NAS search spaces with no change. Third, in Section 3.4.4, we evaluate GenNAS on end-to-end NAS tasks, which outperforms existing works with significant speedup.

Experiment Setup. We consider 13 CNN NAS search spaces including NASBench-101 [73], NASBench-201 [77], Network Design Spaces (NDS) [97], and one NLP search space, NASBench-NLP [72]. All the training is conducted using only one batch of data with batch size 16 for 100 iterations. Details of NAS search spaces and experiment settings are in the supplemental material.

3.4.1 Effectiveness of Synthetic Signals

The synthetic signal analysis is performed on NASBench-101 using the CIFAR-10 dataset. From the whole NAS search space, 500 network architectures are randomly sampled with a known quality ranking provided by NASBench-101. We train the 500 networks using different synthetic signal tensors and calculate their ranking correlations with respect to the groundtruth ranking. Using the CNN architecture discussed in Section 3.3.1, we consider three stages, S_1 to S_3 for $N = 3$; the number of channels is 64 for each stage. For `Sin1D` and `Sin2D`, we set three ranges for frequency f: low (L) $f \in (0, 0.125)$, medium (M) $f \in (0.125, 0.375)$, and high (H) $f \in (0.375, 0.5)$. Within each frequency range, 10 signals are generated using uniformly sampled frequencies. For `Dot` and `GDot`, we randomly set 50% and 100% pixels to ± 1 on the

Table 3.1: Ranking correlation (Spearman's ρ) analysis of different synthetic signals on NASBench-101.

Stage	Sin1D			Sin2D			Dot		GDot		Resize	Zero
	L	M	H	L	M	H	50%	100%	50%	100%		
S_1	0.13	0.43	**0.64**	0.14	0.53	0.63	0.55	0.62	0.18	0.16	0.56	0.17
S_2	0.03	0.52	**0.79**	0.05	0.73	0.72	0.64	0.69	0.03	0.02	0.73	0.18
S_3	0.08	0.77	0.80	0.23	0.78	0.72	0.76	**0.81**	0.16	0.17	0.80	0.22
GenNAS-combo:: 0.85												

zeroized feature maps.

The results of ranking correlations are shown in Table 3.1. The three stages are evaluated independently and then used together. Among the three stages, `Sin1D` and `Sin2D` within medium and high frequency work better in S_1 and S_2, while the high frequency `Dot` and `resize` work better in S_3. The low-frequency signals, such as `GDot`, `Sin1D`-L, `Sin2D`-L, and the extreme case `zero` tensors, result in low ranking correlations; we attribute to their poor distinguishing ability. We also observe that the best task in S_3 (0.81) achieves higher ρ than S_1 (0.64) and S_2 (0.79), which is consistent with the intuition that the features learned in deeper stages have more impact to the final network quality.

When all three stages are used, where each stage uses its top-3 signal bases, the ranking correlation can achieve 0.85, higher than the individual stages. This supports our assumption in Section 3.3.1 that utilizing more intermediate information of a network is beneficial. From this analysis, we choose two top-performing proxy tasks in the following evaluations to demonstrate the effectiveness of regression: **GenNAS-single** – the best proxy task with a single signal tensor `Dot 100%` used only in S_3, and **GenNAS-combo** – the combination of the three top-performing tasks in three stages.

3.4.2 Effectiveness and Efficiency of Regression without Proxy Task Search

To quantify how well the proposed regression can approximate the neural architecture quality with only one batch of data within seconds, we use the ranking correlation, Spearman's ρ, as the metric [19, 85, 76]. We use the two manually designed proxy tasks (GenNAS-single and GenNAS-combo) without proxy task search to demonstrate that **GenNAS is generic and can be directly applied to any new search spaces with zero task-**

specific search efforts. The evaluation is extensively conducted on 13 CNN search spaces and 1 NLP search space, and the results are summarized in Table 3.2.

On NASBench-101, GenNAS is compared with zero-cost NAS [19, 79] and the latest classification based approaches [85]. Specifically, NASWOT [79] is a zero-training approach that predicts a network's trained accuracy from its initial state by examining the overlap of activations between datapoints. Abdelfattah et al. [19] proposed proxies such as synflow to evaluate the networks, where the synflow computes the summation of all the weights multiplied by their gradients and has the best-reported quality in the paper. Liu et al. [85] used three unsupervised classification training proxies, namely rotation prediction (rot), colorization (col), and solving jigsaw puzzles (jig), and one supervised classification proxy (cls). We report their results after

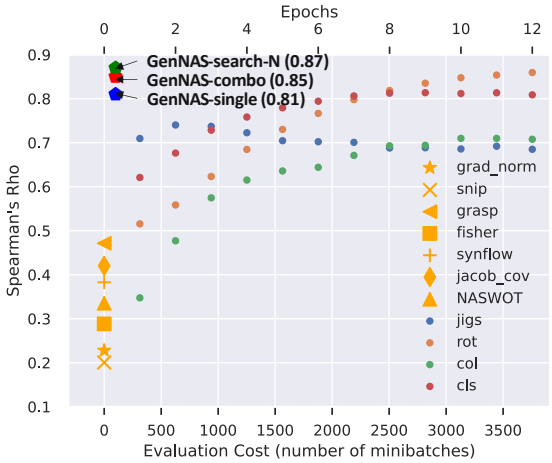


Figure 3.5: The effectiveness of regression-based proxy task. GenNAS significantly outperforms all the existing NAS evaluation approaches regarding ranking correlation, with near-zero training cost.

10 epochs (@ep10) for each proxy. The results show that GenNAS-single and GenNAS-combo achieve 0.81 and 0.85 ρ on CIFAR-10, and achive 0.73 on ImageNet, respectively, much higher than NASWOT and synflow. It is also comparable and even higher comparing with the classification proxies, cls@ep5 and cls@ep10. Notably, the classification proxies need to train for 10 epochs using *all training data*, while GenNAS requires only a few seconds, more than 40× faster. On NASBench-201, we further compare with vote [19] and EcoNAS [69]. EcoNAS is a recently proposed reduced-

training proxy NAS approach. Vote [19] adopts the majority vote between three zero-training proxies including synflow, jacob_cov, and snip. Clearly, GenNAS-combo outperforms all these methods regarding ranking correlation and is also 60× faster than EcoNAS and 40× faster than cls@ep10. On Neural Design Spaces, we evaluate GenNAS on both CIFAR-10 and ImageNet datasets. Comparing with NASWOT and synflow, GenNAS-single and GenNAS-combo achieve higher ρ in almost all cases. Also, synflow performs poorly on most of the NDS search spaces especially on ImageNet dataset, while GenNAS achieves even higher ρ. Extending to NLP search space, NASBench-NLP, GenNAS-single and GenNAS-combo achieve 0.73 and 0.74 ρ, respectively, surpassing the best zero-proxy method (0.56). Comparing with the ppl@ep3, the architectures trained on PTB [98] dataset after three epochs, GenNAS is 192× faster in prediction.

Figure 3.5 visualizes the comparisons between GenNAS and existing NAS approaches on NASBench-101, CIFAR-10. Clearly, regression-based GenNAS (single, combo) significantly outperforms the existing NAS with near-zero training cost, showing remarkable effectiveness and high efficiency.

3.4.3 Effectiveness of Proxy Task Search and Transferability

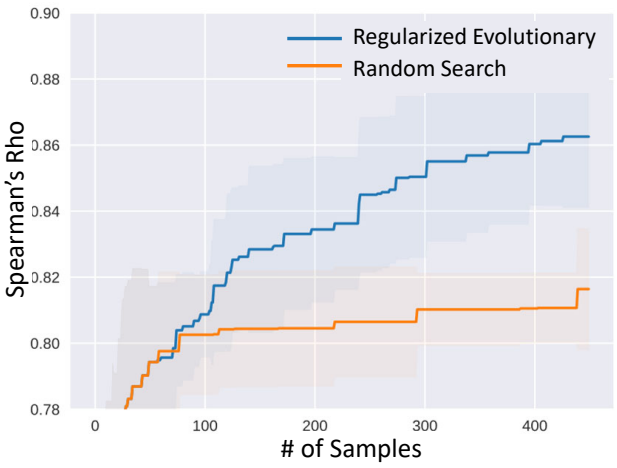


Figure 3.6: Proxy task search.

Effectiveness of Proxy Task Search. While the *unsearched* proxy tasks can already significantly outperform all existing approaches (shown

in Section 3.4.2), we demonstrate that the proxy task search described in Section 3.3.2 can further improve the ranking correlation. We adopt the regularized evolutionary algorithm [31]. The population size is 50; the tournament sample size is 10; the search runs 400 iterations. We randomly select 20 architectures with groundtruth for calculating the ρ. More settings can be found in the supplemental material. Figure 3.6 shows the search results averaged from 10 runs with different seeds. It shows that the regularized evolutionary algorithm is more effective comparing with random search, where the correlations of 20 architectures are 0.86 ± 0.02 and 0.82 ± 0.01, respectively.

In the following experiments, we evaluate three searched proxy tasks, denoted by **GenNAS search-N**, **-D**, and **-R**, meaning that the task is searched on NASBench-101, NDS DARTS search space, and NDS ResNet search space, respectively. We study the quality and transferability of the searched tasks on all NAS search spaces. Proxy task search is done on a single GPU GTX 1080Ti. On NASBench-101 (**GenNAS-N**), ResNeXt (**GenNAS-R**), and DARTS (**GenNAS-D**), the search time is 5.75, 4, and 12.25 GPU hours, respectively. Once the proxy tasks is searched, it can be used to evaluate any architectures in the target search space and can be transferred to other search spaces.

GenNAS with Searched/Transferred Proxy Task. The quality of GenNAS-search-N, -D, and -R proxy tasks is shown in Table 3.2. First, in most cases, proxy task search improves the ranking correlation. For example, in NDS, using the proxy task searched on DARTS space (search-D) outperforms other GenNAS settings on DARTS-like spaces, while using proxy task search-R on ResNet-like spaces outperforms others as well. In NASBench-NLP, the proxy task search can push the ranking correlation to 0.81, surpassing the ppl@ep3 (0.79). Such results demonstrate the effectiveness of our proposed proxy task search. Second, the searched proxy task shows great transferability: the proxy task searched on NASBench-101 (search-N) generally works well for other search spaces, i.e., NASBench-201, NDS, and NASBench-NLP. This further emphasizes that the fundamental principles for top-performing neural architectures are similar across different tasks and datasets. Figure 3.5 visualizes the quality of GenNAS comparing with others.

3.4.4 GenNAS for End-to-End NAS

Finally, we evaluate GenNAS on the end-to-end NAS tasks, aiming to find the best neural architectures within the search space. Table 3.3 summarizes the comparisons with the state-of-the-art NAS approaches, including previously used NASWOT, synflow, cls@ep10, and additionally Halfway [73], RSPS [71], DARTS-V1 [18], DARTS-V2, GDAS [99], SETN [100], and ENAS [101]. Halfway is the NASBench-101-released result using half of the total epochs for network training. In all the searches during NAS, we do not use any tricks such as warmup selection [19] or groundtruth query to compensate the low rank correlations. We fairly use a simple yet effective regularized evolutionary algorithm [31] and adopt the proposed regression loss as the fitness function. The population size is 50, and the tournament sample size is 10 with 400 iterations. On NASBench-101, GenNAS finds better architectures than NASWOT and Halfway while being up to 200× faster. On NASBench-201, GenNAS finds better architectures than the state-of-the-art GDAS within 0.3 GPU hours, being 30× faster. Note that GenNAS uses the proxy task searched on NASBench-101 and transferred to NASBench-201, demonstrating remarkable transferability. On Neural Design Spaces, GenNAS finds better architectures than the cls@ep10 using labeled classification while being 40× faster. On NASBench-NLP, GenNAS finds architectures that achieve 0.246 (the lower the better) average final regret score r, outperforming the ppl@ep3 (0.268) with 192× speed up. The regret score r at the moment t is $r(t) = L(t) - L^\star$, where $L(t)$ is the testing log perplexity of the best found architecture according to the prediction by the moment, and $L^\star = 4.36$ is the lowest testing log perplexity in the whole dataset achieved by LSTM [92] architecture.

On DARTS search space, we also perform the end-to-end search on ImageNet-1k [102] dataset. We fix the depth (layer) of the networks to be 14 and adjust the width (channel) so that the # of FLOPs is between 500M to 600M. We evaluate two strategies: one without task search using GenNAS-combo (see Table 3.1), and the other GenNAS-D14 with proxy task search on DARTS search space with depth 14 and initial channel 16. The training settings follow TENAS [103]. The results are shown in Table 3.4. We achieve top-1/5 test error of 25.1/7.8 using GenNAS-combo and top-1/5 test error of 24.3/7.2 using GenNAS-D14, which are on par with existing NAS architectures. GenNAS-

combo is much faster than existing works, while GenNAS-D14 pays extra search time cost. Our next step is to investigate the searched tasks and demonstrate the generalization and transferrability of those searched tasks to further reduce the extra search time cost.

These end-to-end NAS experiments strongly suggest that GenNAS is generically efficient and effective across different search spaces and datasets.

Table 3.2: GenNAS ranking correlation evaluation using the correlation Spearman's ρ. **GenNAS-single** and **GenNAS-combo** use a single or a combination of synthetic signals that are manually designed *without proxy task search*. **GenNAS search-N, -D, -R** mean the proxy task is searched on NASBench-101, NDS DARTS design space, and NDS ResNet design space, respectively. The top-1/2/3 results of GenNAS and efficient NAS baselines are highlighted by $^{\dagger}/^{\ddagger}/^{\S}$ respectively for each task. The values with superscripts are obtained after task search (^{s}) or transferred (^{t}) from a previous searched task. Methods like jig@ep10 which is 40x slower compared to the GenNAS in prediction are not considered as efficient ones.

NASBench-101

Dataset	NASWOT [79]	synflow [19]	jig@ep10	rot@ep10	col@ep10	cls@ep10	GenNAS		
			> 40× slower				single	combo	search-N
CIFAR-10	0.34	0.38	0.69	0.85	0.71	0.81	$0.81^{\S}$	$0.85^{\ddagger}$	$\mathbf{0.87}^{\dagger s}$
ImgNet	0.21	0.09	0.72	0.82	0.67	0.79	$0.65^{\S}$	$\mathbf{0.73}^{\dagger}$	$0.71^{\ddagger t}$

NASBench-201

Dataset	NASWOT	synflow	jacob_cov	snip	cls@ep10	vote	EcoNAS	GenNAS		
					> 40× slower		> 60× slower	single	combo	search-N
CIFAR-10	$0.79^{\S}$	0.72	0.76	0.57	0.75	0.81	0.81	0.77	$0.87^{\ddagger}$	$\mathbf{0.90}^{\dagger t}$
CIFAR-100	0.81	0.76	0.70	0.61	0.75	$0.83^{\ddagger}$	0.75	0.69	$0.82^{\S}$	$\mathbf{0.84}^{\dagger t}$
ImgNet16	0.78	0.73	0.73	0.59	0.68	$0.81^{\S}$	0.77	0.70	$0.81^{\ddagger}$	$\mathbf{0.87}^{\dagger t}$

Neural Design Spaces

Dataset	NAS-Space	NASWOT	synflow	cls@ep10	GenNAS				
				> 40× slower	single	combo	search-N	search-D	search-R
CIFAR-10	DARTS	0.65	0.41	0.63	0.43	0.68	$0.71^{\S t}$	$\mathbf{0.86}^{\dagger s}$	$0.82^{\ddagger t}$
	DARTS-f	0.31	0.09	0.82	0.51	$\mathbf{0.59}^{\dagger}$	$0.53^{\S t}$	$0.58^{\ddagger t}$	0.52^{t}
	Amoeba	0.33	0.06	0.67	0.52	0.64	$0.68^{\S t}$	$\mathbf{0.78}^{\dagger t}$	$0.72^{\ddagger t}$
	ENAS	0.55	0.19	0.66	0.56	$0.70^{\S}$	0.67^{t}	$\mathbf{0.82}^{\dagger t}$	$0.78^{\ddagger t}$
	ENAS-f	0.43	0.26	0.86	0.65	0.65	$0.67^{\S t}$	$\mathbf{0.73}^{\dagger t}$	$0.67^{\ddagger t}$
	NASNet	0.40	0.00	0.64	0.56	$0.66^{\S}$	0.65^{t}	$\mathbf{0.77}^{\dagger t}$	$0.71^{\ddagger t}$
	PNAS	0.51	0.26	0.50	0.32	0.58	$0.59^{\S t}$	$\mathbf{0.76}^{\dagger t}$	$0.71^{\ddagger t}$
	PNAS-f	0.10	0.32	0.85	0.45	$0.48^{\S}$	$\mathbf{0.56}^{\dagger t}$	$0.55^{\ddagger t}$	0.47^{t}
	ResNet	0.26	0.22	0.65	0.34	0.52	$0.55^{\S t}$	$0.54^{\ddagger t}$	$\mathbf{0.83}^{\dagger s}$
	ResNeXt-A	$0.65^{\S}$	0.48	0.86	0.57	0.61	$0.80^{\ddagger t}$	0.63^{t}	$\mathbf{0.84}^{\dagger t}$
	ResNeXt-B	$0.60^{\S}$	$0.60^{\ddagger}$	0.66	0.26	0.30	0.53^{t}	0.55^{t}	$\mathbf{0.71}^{\dagger t}$
ImageNet	DARTS	0.66	0.21	–	0.61	$0.75^{\ddagger}$	$0.70^{\S t}$	$\mathbf{0.84}^{\dagger t}$	0.55^{t}
	DARTS-f	0.20	0.37	–	0.68§	$0.69^{\ddagger}$	0.67^{t}	$\mathbf{0.69}^{\dagger t}$	0.59^{t}
	Amoeba	0.42	0.25	–	0.63	$0.72^{\S}$	$0.73^{\ddagger t}$	$\mathbf{0.80}^{\dagger t}$	0.67^{t}
	ENAS	$0.69^{\S}$	0.17	–	0.59	$0.70^{\ddagger}$	0.58^{t}	$\mathbf{0.81}^{\dagger t}$	0.65^{t}
	NASNet	0.51	0.01	–	0.52	$0.59^{\S}$	0.52^{t}	$\mathbf{0.70}^{\dagger t}$	$0.61^{\ddagger t}$
	PNAS	$0.60^{\ddagger}$	0.14	–	0.28	0.39	$0.45^{\S t}$	$\mathbf{0.62}^{\dagger t}$	0.40^{t}
	ResNeXt-A	0.72	0.42	–	$0.80^{\S}$	$0.84^{\ddagger}$	0.75^{t}	0.62^{t}	$\mathbf{0.87}^{\dagger t}$
	ResNeXt-B	0.63	0.31	–	$0.71^{\ddagger}$	$\mathbf{0.79}^{\dagger}$	0.51^{t}	0.60^{t}	$0.64^{\S t}$

NASBench-NLP

Dataset	grad_norm	snip	grasp	synflow	jacob_cov	ppl@ep3	GenNAS		
						> 192× slower	single	combo	search
PTB	0.21	0.19	0.16	0.34	$0.56^{\S}$	0.79	0.73	$0.74^{\ddagger}$	$\mathbf{0.81}^{\dagger s}$

Table 3.3: GenNAS end-to-end NAS results comparing with the state-of-the-art NAS approaches, showing test accuracy (%) on different NAS-spaces and datasets. * denotes a method that is replicated with the same regularized evolutionary algorithm in Section 3.4.4 for fair comparison. On NASBench-201, the GPU hours do not include task search since GenNAS-N is transferred from NASBench-101. The values with superscripts are obtained after task search (^{s}) or transferred (^{t}) from a previous searched task.

NASBench-101(CIFAR-10)								
Optimal	NASWOT*	synflow*	Halfway*	**GenNAS-N**				
94.32	93.30±0.002	91.31±0.02	93.28±0.002	**93.92±0.004**^{s}				
NASBench-201								
Dataset	Optimal	RSPS	DARTS-V2	GDAS	SETN	ENAS	NASWOT	**GenNAS-N**
CIFAR-10	94.37	84.07±3.61	54.30±0.00	93.61±0.09	87.64±0.00	53.89±0.58	92.96±0.80	**94.18±0.10**^{t}
CIFAR-100	73.49	58.33±4.34	15.61±0.00	70.61±0.26	56.87±7.77	15.61±0.00	70.03±1.16	**72.56±0.74**^{t}
ImgNet16	47.31	26.28±3.09	16.32±0.00	41.71±0.98	32.52±0.21	14.84±2.10	44.43±2.07	**45.59±0.54**^{t}
GPU hours		2.2	9.9	8.8	9.5	3.9	0.1	0.3
Neural Design Spaces (CIFAR-10)								
NAS-Space	Optimal	NASWOT*	synflow*	cls@ep10*	**GenNAS-N**	**GenNAS-R**	**GenNAS-D**	
ResNet	95.30	92.81±0.10	93.52±0.31	94.51±0.20	94.48±0.24^t	94.63±0.23^t	**94.77±0.13**^{s}	
ResNeXt-A	94.99	93.39±0.67	94.05±0.48	94.24± 0.22	94.25±0.21^t	94.12±0.20^t	**94.37±0.14**^{t}	
ResNeXt-B	95.12	93.56±0.33	93.65±0.64	94.33±0.26	**94.29±0.24**^{t}	94.26±0.35^t	94.23±0.32^t	

Table 3.4: Comparisons with state-of-the-art NAS methods on ImageNet under the mobile setting. * is the time for proxy task search.

Method	Test Err. (%)		Params	FLOPS(M)	Search Cost	Searched	Searched
	top-1	top-5	(M)	(M)	(GPU days)	Method	dataset
NASNet-A [29]	26.0	8.4	5.3	564	2000	RL	CIFAR-10
AmoebaNet-C [31]	24.3	7.6	6.4	570	3150	evolution	CIFAR-10
PNAS [104]	25.8	8.1	5.1	588	225	SMBO	CIFAR-10
DARTS(2nd order) [18]	26.7	8.7	4.7	574	4.0	gradient-based	CIFAR-10
SNAS [105]	27.3	9.2	4.3	522	1.5	gradient-based	CIFAR-10
GDAS [99]	26.0	8.5	5.3	581	0.21	gradient-based	CIFAR-10
P-DARTS [106]	24.4	7.4	4.9	557	0.3	gradient-based	CIFAR-10
P-DARTS	24.7	7.5	5.1	577	0.3	gradient-based	CIFAR-100
PC-DARTS [74]	25.1	7.8	5.3	586	0.1	gradient-based	CIFAR-10
TE-NAS [103]	26.2	8.3	6.3	-	0.05	training-free	CIFAR-10
PC-DARTS	24.2	7.3	5.3	597	3.8	gradient-based	ImageNet
ProxylessNAS [27]	24.9	7.5	7.1	465	8.3	gradient-based	ImageNet
UNNAS-jig [85]	24.1	-	5.2	567	2	gradient-based	ImageNet
TE-NAS	24.5	7.5	5.4	599	0.17	training-free	ImageNet
GenNAS-combo	25.1	7.8	4.8	559	0.04	evolution+few-shot	CIFAR-10
GenNAS-D14	24.3	7.2	5.3	599	0.7*+0.04	evolution+few-shot	CIFAR-10

3.5 Summary

In this chapter, we discuss GenNAS, a self-supervised regression-based approach for neural architecture training, evaluation, and search. GenNAS successfully answered the two questions at the beginning. (1) GenNAS *is* a generic task-agnostic method, using synthetic signals to capture neural networks' fundamental learning capability without specific downstream task knowledge. (2) GenNAS *is* an extremely effective and efficient method using regression, fully utilizing all the training samples and better capturing valued information. We show the true power of self-supervised regression via manually designed proxy tasks that do not need to search. With proxy search, GenNAS can deliver even better results. Extensive experiments confirmed that GenNAS is able to deliver state-of-the-art quality with near-zero search time, in terms of both ranking correlation and the end-to-end NAS tasks with great generalizability.

CHAPTER 4

EXTENSIBLE AND EFFICIENT PROXY FOR NEURAL ARCHITECTURE SEARCH

In the previous chapter, we discussed how GenNAS can be deployed on complex search spaces and achieve a certain level of search space/task agnosticism. However, when exploring search spaces intricate or unseen, GenNAS still encounters drawbacks, such as inefficient evaluation times and limited effectiveness in multi-task scenarios.

To address the demanding computational challenges of NAS in designing deep neural networks (DNNs), efficient or near-zero-cost proxies have been recently proposed. These proxies enable the evaluation of candidate architectures with just one iteration of backpropagation, using the obtained scores to predict architecture quality for downstream tasks.

Despite their potential, two significant drawbacks hinder the wide adoption of these efficient proxies: (1) they are not adaptive to various NAS search spaces; and (2) they are not extensible to multi-modality downstream tasks.

To address these two issues, we first propose an Extensible proxy (Eproxy) that utilizes self-supervised, few-shot training to achieve near-zero costs. A crucial component of Eproxy's efficiency is the introduction of a "barrier layer" with randomly initialized frozen convolution parameters, which adds non-linearities to the optimization spaces so that Eproxy can discriminate the quality of architectures at an early stage. We further propose a Discrete Proxy Search (DPS) method to find the optimized training settings for Eproxy with only a handful of benchmarked architectures on the target tasks. Our extensive experiments confirm the effectiveness of both Eproxy and DPS. On the NDS-ImageNet search spaces, Eproxy+DPS achieves a higher average ranking correlation (Spearman $\rho = 0.73$) than the previous efficient proxy (Spearman $\rho = 0.56$). On the NAS-Bench-Trans-Micro search spaces with seven tasks, Eproxy+DPS delivers comparable quality with the early stopping method (146× faster). For the end-to-end task such as DARTS-ImageNet-1k, our method delivers better results than NAS per-

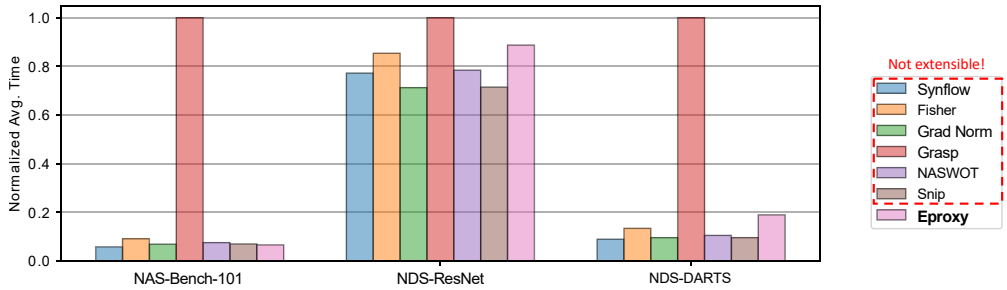


Figure 4.1: Eproxy is compared to six efficient proxies in terms of evaluation speed on NAS-Bench-101, NDS-ResNet, and NDS-DARTS search spaces, demonstrating that Eproxy falls within the zero-cost category. The plot shows the normalized average time.

formed on CIFAR-10 while only requiring one GPU hour with a single batch of CIFAR-10 images.

4.1 Introduction

As deep neural networks (DNNs) find uses in a wide range of applications, such as computer vision [1, 2, 62, 107] and natural language processing [5, 108, 92, 109, 7], Neural Architecture Search (NAS) [29, 31, 34, 110, 18] has become an increasingly important technique to automate the design of neural architectures for different tasks [111, 112, 113, 114]. Recent progress in NAS has demonstrated superior results, surpassing those of human designs [29, 36, 34]. However, one major hurdle for NAS is its high computation cost. For example, the seminal work of NAS [29] consumed 2000 GPU hours to obtain a high-quality DNN, a prohibitively high cost for many researchers. The high computation cost of NAS can be attributed to three major factors: (1) the large search space for candidate neural architectures, (2) the training of various candidate neural architectures, and (3) the comparison of the solution quality of candidate neural architectures to guide the NAS search process. Subsequent NAS work has proposed various techniques to address the above issues, such as the constraints on the search space [54], the weight-sharing networks to reduce the training cost [115], and search with synthetic or random labels [85, 116].

Out of the advancement, the latest efficient proxies showed that the quality of a neural architecture could be determined by a proxy metric computed

within seconds without full training. Hence they are near zero-cost (ZC). The activations of an untrained network were analyzed as a proxy in NAS-WOT [79] with promising results. Abdelfattah et al.[19] suggested various proxies for pruning. Zhang et al.[117] proposed a zero-cost pruning method for differentiable NAS at initialization. ZenNAS [118] quantifies network expressivity, which has a positive correlation with model accuracy. ZiCo [119] leverages gradient properties to improve neural network convergence rate. The efficient proxies discussed above have, however, two primary drawbacks:

1. The quality of these efficient proxies is inconsistent across different search spaces. While many proxies show high correlations in the confined search spaces of small NAS Benchmarks, their quality can be vastly different in real-world applications where the search spaces are orders of magnitude larger than those of the tabular benchmarks. For instance, Synflow demonstrates a high ranking correlation on NAS-Bench-201 [120], but its quality is subpar on NAS-Bench-101 [73], which is 27 times larger than NAS-Bench-201.

2. Efficient proxies don't adapt well to multi-modality downstream tasks. These proxies are typically tailored for CIFAR-10 like classification tasks, resulting in promising prediction results for these specific tasks. However, their quality drops significantly in other scenarios. As an illustration, NASWOT exhibits a dismal average ranking correlation of 0.03 on NAS-Bench-MR [121], which consists of nine real-world tasks. Many efficient proxies employ specific algorithms, like pruning, to convert the weights of architectures into prediction values. This rigid algorithmic approach hinders the proxy's adaptability to tasks beyond classification. Additionally, some zero-cost proxies have been found to favor certain neural architectures [122]. For example, both empirical and theoretical evidence suggest that Synflow has a preference for large models [123].

We question if it is possible for a zero-cost, few-shot proxy task to accurately mimic the true task and result in similar quality ranks for architectures, as shown in Figure 4.2. This leads us to introduce a novel and efficient proxy, called the Extensible proxy (Eproxy), approached from a different perspective. Recently, in our prior work, Li et al. [124] introduced a method that

employs regression-based training termed GenNAS for the assessment of architectures. Although innovative, their approach is not devoid of costs, both in terms of time and resources. In contrast, our work takes inspiration from the GenNAS framework, achieving nearly zero-cost evaluations. Unlike previous efficient proxies, Eproxy utilizes few-shot spatial-level regression on a set of image-label pairs (see Illustration in Figure 4.4). The labels are 2D synthetic features because, as suggested by Li et al. [124], spatial-level regression is more challenging than one-hot classification on a tiny dataset, i.e., a batch of image-label pairs. The key component of Eproxy is the barrier layer. It takes the output of the architecture network into an untrained convolutional layer and performs the regression with the labels. Such a simple mechanism can significantly improve the quality of Eproxy to identify good architectures and bad ones when performing 10 iterations of backpropagation, i.e., near zero-cost. We find that the barrier layer increases the complexity of the optimization space. Hence, poor-quality architectures are more difficult to optimize (see Section 4.3.1). Since Eproxy is a configurable few-shot trainer, we design a novel search space for Eproxy that includes various hyperparameters, such as feature combinations, output channel numbers, and selection for barrier layers, which makes up Eproxy's multi-modalities. We term such a search method as Discrete Proxy Search (DPS) (the quality of which is shown in Figure 4.3). Notably, besides the evaluation of a handful of architectures' quality, DPS does not need to use any task-specific information (e.g., in our experiment, we only use a single batch of CIFAR-10 [125] images throughout all the experiments).

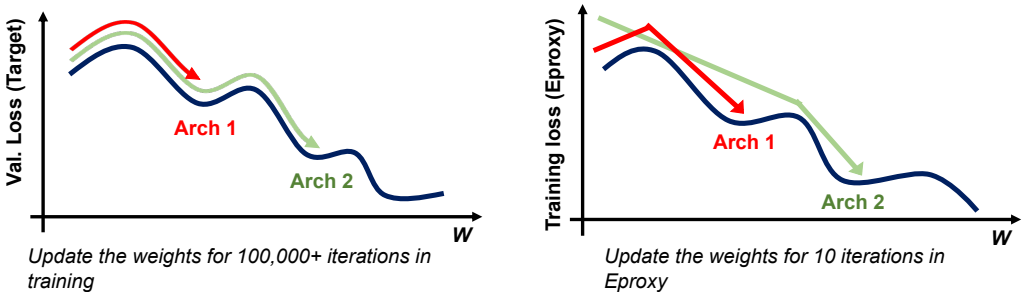
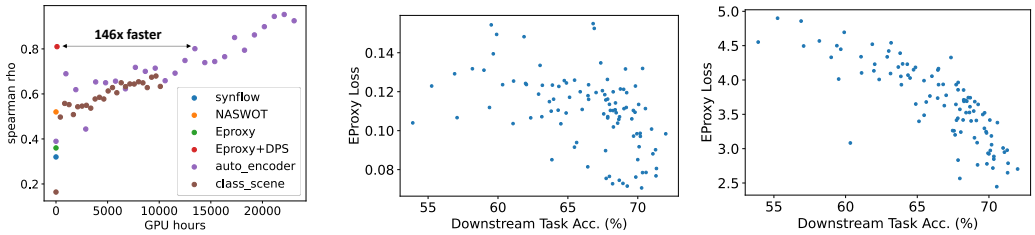


Figure 4.2: A hypothetical illustration of the validation losses of two architectures on a downstream task is shown on the left. The losses of a sophisticated few-shot proxy, on the right, can reflect the actual quality of the architectures on target tasks. The "W" represents updated weights.



(a) Autoencoder: Time vs Corr.

(b) DARTS-ImgNet: Eproxy

(c) DARTS-ImgNet: Eproxy+DPS

Figure 4.3: **a:** Comparison with efficient proxies and early stopping methods on NAS-Bench-Trans-Micro Autoencoder task. It shows the effectiveness of DPS compared with early stopping methods on either the target task or a classification task when evaluating 4096 architectures. **b, c:** On NDS DARTS-**ImageNet** task, Eproxy and Eproxy+DPS (Searched on DARTS-CIFAR-10, transferred to ImageNet) achieve 0.51, 0.85 ρ respectively. It shows DPS can find a search-space-aware Eproxy.

We summarize our contributions as follows:

1. We propose an efficient proxy task with the barrier layer that utilizes a few-shot self-supervised regression. The task adopts one batch of images in the CIFAR-10-level dataset (not necessarily from the target training dataset). It uses synthetic labels to evaluate architectures. Eproxy significantly speeds up the traditional early stopping evaluation process while maintaining the high-ranking correlation.

2. We propose the downstream-task and search-space aware proxy search algorithm with a proxy search space. We formulate the proxy task search as a discrete optimization problem with only a handful of architectures, such that the quality rankings of the networks on the ground-truth task and the proxy task should be consistent. The searched Eproxy can accurately evaluate the quality of network architectures and make Eproxy search-space and downstream-task aware.

3. We provide thorough experiments to evaluate the quality of Eproxy and Eproxy boosted by DPS on more than 30 search spaces and tasks. We demonstrate that our methods have overall higher quality than existing efficient proxies in terms of all three factors: architecture ranking correlation score, top-10%-architecture retrieve rate, and end-to-end NAS quality. Our solid experimental results can be further utilized to

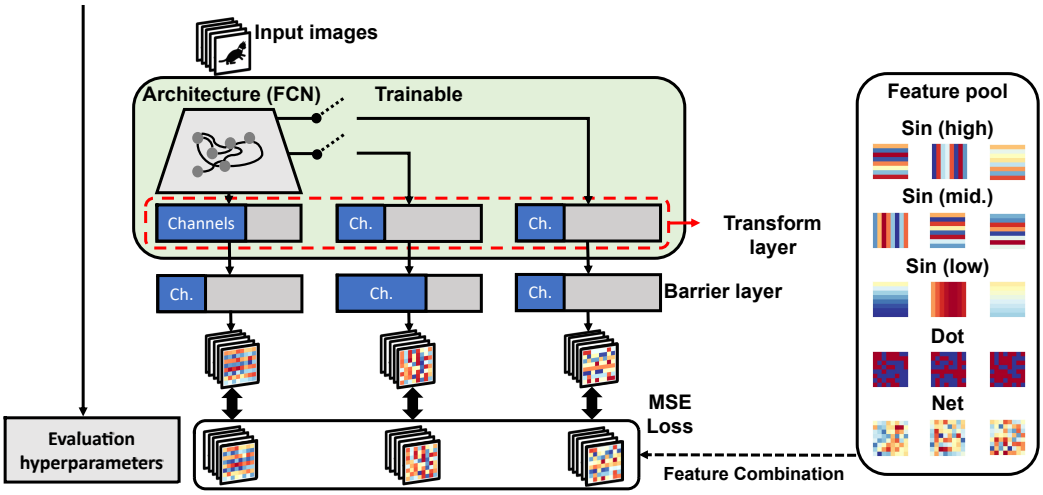


Figure 4.4: The design of Eproxy and the searchable components. Dotted line: The configurable components for Discrete Proxy Search. Green block: Trainable components. DPS can further utilize the configuration of Eproxy to target search spaces and downstream tasks.

benefit the NAS community.

4.2 Our Approaches

In Sec. 4.2.1, we present the Eproxy for efficient network evaluation. Sec. 4.2.2 explains how to find a task and search-space aware Eproxy using Discrete Proxy Search.

4.2.1 Extensible NAS Proxy

The Eproxy is designed for the architectures to learn the output of an untrained network on a set of image-label pairs (See Figure 4.4). We utilize the MSE-based training [124] with a large learning rate and limited backpropagation steps to make it as efficient as the existing near zero-cost proxies. However, directly applying a few-shot regression task with a large learning rate leads to poor correlation based on our observations. To make the Eproxy architecture-quality-aware within a few iterations, we propose an untrained barrier layer to make the task more involved (See Section 4.3.1). The barrier layer is a randomly initialized convolution layer to the output of the trainable components. Our experiments show that adding such a layer can

significantly improve the correlation between the predictions and the quality of neural architectures in the downstream tasks within a few back propagations (Sec 4.3.1). To be more specific, the Eproxy training loss can be described as:

$$\min_{w_a, w_t} \mathcal{L}_{MSE}(G(w_b, F(w_a, w_t, X)), Y) \tag{4.1}$$

where the $X \in \mathbb{R}^{b \times c_{in} \times h_{in} \times w_{in}}$ is the a set of input images (b is batch size; c_{in} is number of input channels). F is a fully convolutional neural network (FCN) with a transform layer (a convolutional layer) that transforms the X to $F(\cdot) \in \mathbb{R}^{b \times c_{mid} \times h_{out} \times w_{out}}$. The FCN is obtained using the architecture without a task-specified head in the downstream tasks. For example, the classifier network with the classification (last pooling and linear layer) head was removed. w_a and w_t are the weights of architecture for evaluation and the weights of the transform layer that project the output channels of the architecture to c_{mid} which is the number of the transform layer's output channels. G is the barrier layer, and w_b is the weights. Note in the Eproxy without DPS, $Y \in \mathbb{R}^{b \times c_{out} \times h_{out} \times w_{out}}$ is the output of an untrained 6-layer CNN (Figure 4.4, 'Net').

We further provide the pseudo-code of Eproxy in Listing 4.1. The Eproxy utilizes a batch of images and corresponding synthetic features as labels (configurations such as the combination of features and output channel numbers can be searched by DPS). The model is trained for 10 iterations. The model's quality is gauged by the MSE loss (lower values denoting better quality).

```
def Eproxy(arch, barrier, img, features, t_iter = 10, **
    kwargs):
    # img shape: B, C = 3, W_in, H_in
    # features shape: B,  C_out, W_out, H_out
    optimizer = SGD(arch.parameters())
    for i in range(t_iter):
        output_mid = model(img) # B, C_mid, W_out, H_out
        output = barrier(output_mid) # B, C_out, W_out, H_out
        loss = ((output - features)**2).mean()
        optimizer.zero_grad()
        loss_m.backward()
        optimizer.step()
    return loss
```

Listing 4.1: Pseudo PyTorch-style code for Eproxy.

4.2.2 Discrete Proxy Search

Since Eproxy provides abundant configurable hyperparameters and utilizes data-agnostic spatial labels, different settings can be naturally adjusted for tasks and search spaces. Therefore, we propose a semi-supervised discrete proxy search to find a setting that can be suitable for the specific modality. As shown in Figure 4.4, the searchable configurations are provided as follows:

1. Transform and barrier layer: Both layers can have kernel size selected from $\{1, 3, 7\}$, and the channel number c_{mid} can be selected from 16 to 512 geometrically with 2 as a multiplier.

2. Feature combination: a) Untrained CNN outputs. The experiment results show that an untrained network's output features can be powerful for evaluating architectures on numerous tasks/search spaces. The synthetic features can be interpreted as a tiny knowledge-distillation task from an untrained teacher network. b) Sine wave: we adopt the sine wave features with low/mid/high frequency along width/height. The insight is that good CNNs can learn different frequency signal [124, 94]. The features are generated by $\sin(2\pi fx + \phi)$ or $\sin(2\pi fy + \phi)$ with equal probability to be selected. We set three ranges for frequency f: low (L) $f \in (0, 0.125)$, medium (M) $f \in (0.125, 0.375)$, and high (H) $f \in (0.375, 0.5)$. c) Dot: By utilizing the Rademacher distribution, we generate the synthetic features with only ± 1. The features attempt to simulate the spatial classification that is widely adopted in tasks such as detection [126], segmentation [127], tracking [127, 128]. The combined features can be multiplied by an augment coefficient selected from 0.5 to 2 with 0.5 as a step.

3. Training hyperparameters: a) Learning rate: we adopt the SGD optimizer, and the learning rate can be selected from 0.5 to 1.5 with 0.1 as the step. b) Initialization: we adopt two initialization methods, Kaiming [129] and Xavier [130], with either Gaussian or Uniform initialization, resulting a total of four choices.

4. Intermediate output evaluation: We provide the choices to force the network to learn the intermediate outputs from the layer before the first or second downsample layer. The motivation is that earlier stages of the

network have different learning behaviors from the deeper stages [131]. Thus, monitoring the early stages can give more flexibility for adapting Eproxy to different tasks.

5. FLOPS: As works [132, 36, 123, 133] suggested that FLOPS is a good indicator for architecture quality. Hence we incorporate the FLOPS normalized by the largest architecture in the search space with the Eproxy loss as $\mathcal{L} \cdot (1 + \alpha \cdot \text{norm}(\text{FLOPS}))$, where α can be selected from -0.5 to 0.5 with 0.1 steps.

The total number of configuration combinations in the proxy search space is 5×10^{15}. We utilize the regularization evolutionary algorithm (REA) [31] to conduct the exploration. First, we randomly sample a small subset of the neural architectures in the NAS search space and obtain their ground truth ranking on the target task or a highly correlated down-scaled task (for example, CIFAR-10 is a good proxy for ImageNet). We then evaluate these networks using Eproxy with different configurations and calculate the quality ranking correlation ρ of the Eproxy and the target task, and the ρ is the fitness function for REA.

We provide the pseudo code of the DPS function in Listing 4.2. This function utilizes a given set of architectures and their corresponding ground-truth accuracies (`archs_accs`) and iterates over a specified number of cycles (`cycle`). DPS starts with an initial population of configurations, each of which includes parameters like learning rate, channel numbers, feature combinations, among others. The `REAEngine` class is utilized to manage the Regularized Evolutionary Algorithm. It aids in generating random configurations, calculating correlations between Eproxy losses (under the given configuration) and ground-truth accuracies for a set of architectures, and performing evolutionary mutations on the configurations. The function accumulates and returns the history of configurations and their corresponding correlations over the evolutionary cycles in `config_history`.

```
def DPS(archs_accs, cycle, population = 40, sample = 10, **
    kwargs):
     # len(archs_accs): 20 pairs of archs & gt accs.
     # config: including lr, channel number, feature
    combination, etc.
     config_history = []
     rea = REAEngine(population, sample, mutation_rate)
```

```
    # generate the initial pool
    for _ in range(population):
        config = rea.get_random_config()
        corr = rea.get_corr(config, archs_accs) # calculate
    the correlation between Eproxy (under the config) losses
    and gt accs based on 20 archs.
        config_history.append({config: corr})
    # evolution
    for _ in range(cycle):
        new_config = rea.get_mutate_config()
        corr = rea.get_corr(new_config, archs_accs)
        config_history.append({new_config: corr})
    return config_history
```

Listing 4.2: Pseudo PyTorch-sytle code for DPS.

4.3 Experiments

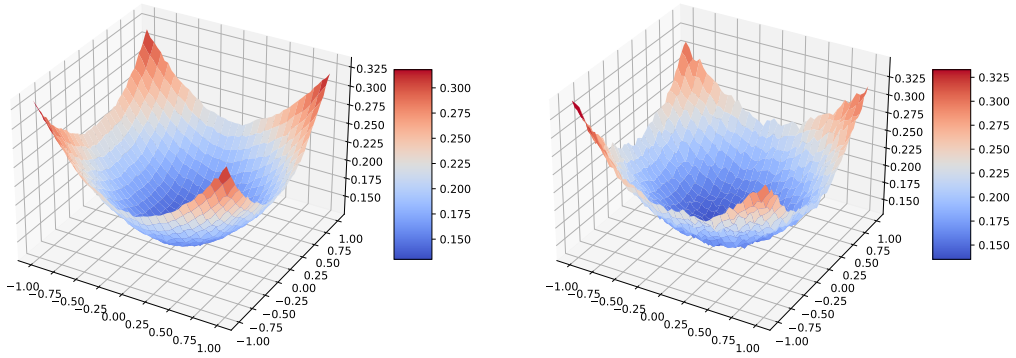
In this section, we perform the following evaluations for Eproxy and DPS. First, in Sec. 4.3.1, we conduct the ablation study on NASBench-101 [73], the first and yet the largest tabular NAS benchmark with over 423k CNN models and training statistics on CIFAR-10. We explain the mechanism behind the barrier layer with empirical results. Furthermore, we compared Eproxy and Eproxy boosted by DPS with existing efficient proxies. Second, from Sec. 4.3.2 to Sec. 4.3.4, we use metrics including ranking correlation, top-10 architecture retrieve rate [134] to evaluate the proposed method on **NDS** [135] (11 search spaces on CIFAR-10, 8 search spaces on ImageNet), **NAS-Bench-Trans-Micro** [136] (7 tasks), and **NAS-Bench-MR** [121] (9 tasks). Third, in Sec. 4.3.5, we evaluate the end-to-end NAS on NAS-Bench-101/201. Moreover, we report the end-to-end search on the DARTS-ImageNet search space in Sec. 4.3.5.

4.3.1 Ablation Study on NAS-Bench-101

We study the effectiveness of our barrier layer in this section. We use the tool from the work [137] to visualize the loss surface of an architecture selected randomly from NAS-Bench-101 on our few-shot regression task. Fig-

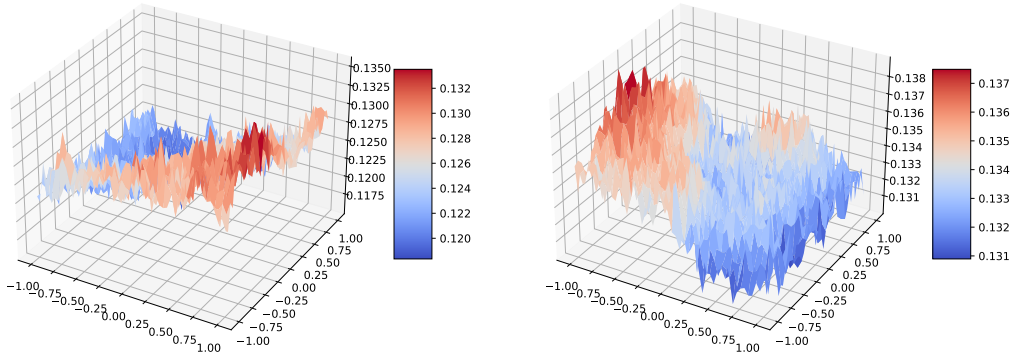
ure 4.5a, 4.5b shows the loss surface without the barrier has a good convexity, which indicates the task is simple, as we use a proxy task that contains very few samples (16 image-label pairs) for a shorter evaluation period. The simplicity of the proxy task gives us two potential problems that can affect the final results. (1) If a task is too simple, every model can perform similarly well. (2) When the optimization is easy, models can have similar quality at the early stage of training. As we observed, loss surfaces from different models have similar shapes without barriers, requiring us to use more training steps to see the difference between good and bad architectures. To mitigate these two problems, Eproxy added a barrier layer which is a random initialized linear/convolution layer with frozen weights. As shown in Figure 4 (b), the loss surface with the barrier has a noticeable non-convexity, which shows the increased complexity of the proxy task, and now it can better reflect the actual quality of architecture (Figure 4.6a, 4.6b). As the irregular shape of the loss surface varies widely from model to model, it helps us better distinguish the model quality at the early stage of training, allowing us to use fewer training steps to speed up the evaluation further. The results in Table 4.1 show that with the barrier layer, Eproxy can reach ρ 0.65 in only 10 iterations with a learning rate of 1, and it also significantly improves the ranking correlation score with more training iterations.

Next, we sample 20 architectures from NAS-Bench-101 and evaluate DPS. We conduct DPS for 200 epochs, and the total run time is $\sim$20 mins on a single A6000 GPU. In Table 4.2, we report the network evaluation results in terms of Spearman's ρ and top-10% network coverage using the proxy task searched by DPS. We evaluated Eproxy against state-of-the-art zero-cost proxies including NASWOT [79], Synflow [19], ZenNAS [118], and ZiCo [119]. Additionally, we considered architecture parameters and FLOPS as important baselines [133]. Eproxy significantly outperforms existing zero-cost proxies by a large margin. For instance, Synflow, scores 0.45, ZiCo scores 0.61, Eproxy scores 0.65 (without DPS), and Eproxy with DPS scores 0.69. In terms of top-10% retrieval rate, Eproxy with DPS retrieves a higher number of architectures than DPS (38% vs 31%). The results support the efficiency and effectiveness of DPS. Meanwhile, Figure 4.1 confirms that using Eproxy achieves a comparable evaluation speed with other efficient proxies.



(a) Best model without barrier. (b) Worst model without barrier.

Figure 4.5: The loss surfaces of best and worst model from NAS-Bench-101 regression task without barrier.



(a) Best model with the barrier. (b) Worst model with the barrier.

Figure 4.6: The loss surfaces of best and worst model from NAS-Bench-101 regression task with barrier.

4.3.2 NDS

The original Network Design Spaces (NDS) work [135] investigates different search spaces. The NDS is perfect for evaluating efficient proxies in more complex search spaces. For example, researchers benchmarked 5,000 architectures on the DARTS search space and over 20,000 on the ResNet search space. We compared our method with existing zero-cost proxies on **11 search spaces** on **CIFAR-10** and **8 search spaces** on **ImageNet** [102]. We show the results in Table 4.3. Compared to Synflow, ZenNAS, and ZiCo, Eproxy (without DPS) delivers better results on both CIFAR-10 and ImageNet search spaces. Eproxy (without DPS) performs similarly to NAS-WOT on both CIFAR-10 and ImageNet search spaces. Boosted by DPS,

Eproxy delivers significantly better results on target CIFAR-10 search spaces with 36% and 52% improvement on ranking correlation and top-10% retrieve rate, respectively. Notably, Eproxy+DPS searched on CIFAR-10 with 20 architectures performs significantly better on **ImageNet** search spaces without any prior knowledge of the dataset. Compared to NWT, Eproxy with DPS gains 30% and 57% on ranking correlation and top-10% retrieve rate, respectively. The ImageNet experiment demonstrates the efficiency by utilizing the architectures trained on the down-scaled dataset (CIFAR-10) for DPS. It's noteworthy that using FLOPS and Params as evaluation metrics for models may not be suitable in search spaces with limited model size variations, such as DARTS-f, PNAS-f, and ENAS-f.

4.3.3 NAS-Bench-Trans-Micro

Previous experiments suggest that DPS can optimize Eproxy across different search spaces. We further evaluate Eproxy and DPS on NAS-Bench-Trans-Micro, a benchmark that contains 4096 architectures across **seven large tasks** from the **Taskonomy** [138] dataset. The tasks include object classification, scene classification, unscrambling the image, and image upscaling. The search space is similar to NAS-Bench-201 but has four operator choices per edge instead of six. We conduct the DPS on each task using only 20 architectures. We do not have any prior knowledge of the tasks besides the 20 architecture's ground truth quality since DPS only utilizes a batch of CIFAR-10 images as input. The results are shown in Table 4.4. Note that though Eproxy underperforms regarding the ranking correlation, it achieves better top-10% retrieve rate compared to other methods. It also tells that the global ranking correlation is not the golden metric for evaluating the quality of proxies since it merely reflects the difference of top architectures. With the help of DPS, the average ranking correlation and top 10% retrieve rate are significantly improved and substantially better than other methods. Compared to the early stopping method, DPS requires 7.6X less regarding GPU hours (>99% time for obtaining the quality of 20 architectures while the DPS only takes 0.5 GPU hour).

4.3.4 NAS-Bench-MR

We applied Eproxy and DPS to a complex search space, NAS-Bench-MR [121], with nine high-resolution tasks such as 3D detection, ImageNet-level classification, segmentation, and video recognition [102, 140, 141, 142]. Approximately 2,500 architectures were benchmarked. Each architecture underwent full training (over 100 epochs) and followed a multi-resolution paradigm, with each network consisting of four stages. Each stage comprised modularized blocks (parallel and fusion modules). Our work is the first to investigate this benchmark with efficient proxies. The results are displayed in Table 4.5. The full training consumption on the Cls-C task was calculated based on the source code [143], which took approximately 4000 GPU hours. Note that NASWOT, which performs well on NAS-Bench-Trans-Micro, delivers poor quality on most tasks, implying the inconsistent quality of current efficient proxies. Also, we observed that classification rankings are inconsistent with other tasks, such as segmentation and 3D detection. Our Eproxy+DPS experiments suggest that with a 20-architecture set, the ranking correlation and top-10% retrieve rate are considerably improved (**+89%/+78%**).

4.3.5 End-to-end NAS with Eproxy

We evaluate Eproxy and DPS on the end-to-end NAS tasks to find efficient architectures in the search space.

On **NAS-Bench-101**, we utilize the Eproxy as the fitness function for Regularized Evolutionary (RE) algorithm. Our results are shown in Table 4.6 compared with NAO [144], Semi-NAS [145], WeakNAS [146]. Note that Eproxy, without any query (near-zero-cost) from the benchmark, can find architectures that are significantly better than current SoTA efficient proxies, Synflow (+ 0.87%) and NASWOT (+3.01%). With 20 architectures for DPS and 40 queries (total of 60) to retrieve the top architectures during RE, Eproxy+DPS achieves better results than existing SoTA predictor-based NAS WeakNAS with 100 queries (+0.23%). Furthermore, we explore the 70 neighbors of the top architectures (a total of 150 queries) and find architectures with an average of 94.23% accuracy. Note that Semi-NAS with 1000 queries can only reach 94.01%. On **NAS-Bench-201**, we perform

the DPS on the CIFAR-10 dataset, and the found proxy is directly transferred to CIFAR-100 and Tiny-ImageNet. We compare with MCTS [147], LaNAS [148], WeakNAS [146]. In Table 4.7, we show that Eproxy+DPS can find optimal global architectures within the RE search history. Compared to RE, which directly queries the benchmark, our approach reduced 7x/32x/9x query times on three datasets. Compared to predictor-based NAS, Eproxy+DPS also requires fewer queries to discover the optimal architectures. Our results offer an exciting yet promising direction besides pure predictor-based NAS.

DARTS-ImageNet search space We conduct end-to-end search on ImageNet-1k dataset within the DARTS search space as defined in [18]. The network depth is 14 blocks. The input channels are 48, and the FLOPS range from 500M to 600M for the searched architectures. We utilize the 20 samples from the NDS-DARTS search space (not the same search space as the target) and conduct DPS on CIFAR-10 for 200 epochs in one GPU hour. Then we perform the NAS by adopting regularized evolutionary algorithm with the loss of the zero-cost proxy as the fitness function in 0.4 GPU hour. We compare our method with (a) existing works on the DARTS search space [18, 139, 99, 106, 74, 103] and (b) works on the similar search spaces [29, 31, 104, 27]. The results are shown in Table 4.8. Eproxy achieves a top-1/5 test error of 25.2%/8.1% using Eproxy with only 0.5 GPU hours for NAS. With DPS, Eproxy explores the architecture with 24.4%/7.3% as a top-1/top5 test error. Eproxy+DPS significantly outperforms existing NAS on CIFAR-10, such as PC-DARTS [74], and achieves a comparable result with NAS on ImageNet, demonstrating Eproxy and DPS's efficiency. By utilizing the existing proxy on another search space, DPS shows the transferability between search spaces.

Table 4.1: Ranking correlation (Spearman's ρ) analysis for different losses on NASBench-101. "LR" stands for learning rate; "NZC" stands for near-zero-cost. The results suggest that regression with barrier and large learning rate can achieve a high ranking correlation in 10 iterations near zero cost.

Loss	MSE w/o Barrier			MSE w/ Barrier		
LR	1	1e-1	1e-2	1	1e-1	1e-2
10 iters$^{\text{NZC}}$	0.08	-0.22	-0.19	**0.65**	0.46	0.09
100 iters	0.07	0.67	0.76	0.65	**0.79**	0.79
200 iters	0.22	0.64	0.66	0.61	**0.83**	0.81

Table 4.2: Comparison with efficient proxies on NAS-Bench-101 using the Spearman ρ and top-10% retrieve rate. Eproxy outperforms the state-of-the-art zero-shot methods, ZiCo and ZenNAS, in terms of *rho* when DPS is not utilized.

	Grad norm	Snip	Grasp	Fisher	Synflow	NASWOT	ZenNAS	ZiCo	Eproxy	Eproxy+DPS
ρ	0.20	0.16	0.45	0.26	0.37	0.40	0.61	0.61	0.65	**0.69**
Top-10%	2%	3%	26%	3%	23%	29%	39%	**45%**	31%	38%

Table 4.3: Comparison with efficient proxies on NDS search spaces. $_{\text{T}}$ denotes the DPS on CIFAR-10 and transferred to ImageNet. Therefore, it does not necessitate knowledge of the ImageNet dataset. When DPS is not employed, Eproxy performs better than ZiCo on 7 out of 11 CIFAR-10 search spaces and all 8 ImageNet search spaces. However, with DPS, Eproxy shows significant superiority over other methods.

CIFAR-10	DARTS	DARTS-f	AMB	ENAS	ENAS-f	NASNet	PNAS	PNAS-f	Res	ResX-A	ResX-B	Avg.
Synflow	0.42 9%	-0.14 5%	-0.10 3%	0.18 6%	-0.30 2%	0.02 7%	0.25 9%	-0.26 4%	0.21 4%	0.47 25%	0.61 29%	0.12 9%
NASWOT	0.65 29%	0.31 8%	0.29 20%	0.54 31%	0.44 28%	0.42 27%	0.50 24%	0.13 6%	0.29 7%	0.64 28%	0.57 21%	0.43 21%
ZenNAS	0.50 14%	0.01 4%	0.05 7%	0.22 10%	0.07 5%	0.07 11%	0.24 12%	0.20 3%	0.23 2%	0.59 35%	**0.66** 32%	0.26 12%
ZiCo	0.49 13%	0.11 5%	0.09 4%	0.29 12%	0.02 11%	0.16 12%	0.26 9%	0.09 3%	0.23 3%	0.54 32%	0.63 **34%**	0.26 13%
Eproxy	0.38 12%	0.34 17%	0.54 13%	0.59 35%	0.48 31%	0.56 28%	0.22 4%	0.24 4%	0.51 36%	0.47 24%	0.19 10%	0.41 19%
Eproxy+DPS	**0.72** 33%	**0.39** **19%**	**0.56** **29%**	**0.63** **36%**	**0.47** **30%**	**0.54** **32%**	**0.60** **35%**	**0.48** **28%**	**0.56** **36%**	**0.65** **32%**	0.60 19%	**0.56** **29%**

ImageNet	DARTS	DARTS-f	Amoeba	ENAS	NASNet	PNAS	ResX-A	ResX-B	Avg.
Synflow	0.21 0%	-0.36 4%	-0.25 0%	0.17 9%	0.01 0%	0.14 9%	0.42 7%	0.31 13%	0.08 6%
NASWOT	0.66 16%	0.20 8%	0.42 33%	0.69 36%	0.51 33%	**0.61** 10%	0.73 30%	0.63 38%	0.56 26%
ZenNAS	0.21 8%	0.13 4%	0.21 0%	0.22 18%	0.06 17%	0.23 9%	0.60 23%	0.45 25%	0.27 13%
ZiCo	0.24 8%	0.01 4%	0.18 0%	0.26 36%	0.12 17%	0.27 9%	0.52 8%	0.40 19%	0.25 13%
Eproxy	0.51 20%	0.31 17%	0.66 60%	0.58 33%	0.56 30%	0.36 33%	0.73 55%	0.70 **43%**	0.55 36%
Eproxy+DPS$_{\text{T}}$	**0.85** 50%	**0.53** **28%**	**0.66** **60%**	**0.79** 33%	**0.85** 32%	0.60 **35%**	**0.83** **55%**	**0.72** 36%	**0.73** **41%**

Table 4.4: Comparison with efficient proxies and the early stopping method on TransNAS-Bench-Micro. In 5 out of 7 tasks, Eproxy surpasses ZiCo, a recent zero-shot method. Eproxy+DPS outperforms efficient proxies, and the early stopping method which requires 600 GPU hours per task.

	Cls. Scene	Cls Obj	Room Layout	Jigsaw	Seg	Normal	AE	Avg.
Synflow	0.46/16%	0.50/16%	0.45/28%	0.49/19%	0.32/3%	0.52/19%	0.52/34%	0.47/19%
NASWOT	0.57/21%	0.53/21%	0.30/2%	0.41/11%	0.52/30%	0.59/30%	-0.02/2%	0.41/17%
ZenNAS	0.57/**48%**	0.34/32%	0.24/**27%**	0.35/35%	0.37/32%	0.56/47%	0.15/23%	0.37/35%
ZiCo	0.30/31%	0.03/16%	0.04/23%	0.15/26%	0.07/16%	0.29/27%	0.19/9%	0.15/21%
Eproxy	0.15/14%	0.45/34%	0.06/8%	0.17/33%	0.36/46%	0.25/38%	0.61/**80%**	0.29/36%
Eproxy + DPS	0.70/30%	0.56/**44%**	**0.56**/13%	0.64/**45%**	**0.81/53%**	**0.81/63%**	**0.80**/74%	**0.69/46%**
$ES_{\sim \text{660GPU hrs/task}}$	**0.73**/25%	0.01/7%	0.15/7%	**0.74**/21%	0.39/7%	0.65/27%	0.35/11%	0.43/15%

Table 4.5: Comparison with efficient proxies, and Cls-C full training which requires 4000 GPU hours on NAS-Bench-MR. In 9 tasks, Eproxy achieves superior top-10% retrieval rates compared to recent zero-cost methods like ZenNAS and ZiCo. Eproxy + DPS demonstrates outstanding quality, even when compared to full training on Cls-C.

	Cls-A	Cls-B	Cls-C	Cls-10c	Seg	Seg-4x	3dDet	Video	Video-p	Avg.
Synflow	0.25 11%	0.05 14%	0.37 20%	0.21 15%	0.43 17%	0.22 9%	0.22 8%	0.45 18%	0.52 17%	0.30 14.3%
NASWOT	0.37 18%	-0.20 4%	-0.15 2%	-0.39 0%	0.50 10%	0.38 8%	0.48 10%	-0.36 1%	-0.36 0%	0.03 6%
ZenNAS	0.41 4%	0.50 0%	0.30 1%	0.25 0%	0.49 6%	0.22 2%	0.26 10%	0.41 0%	0.39 0%	0.36 3%
ZiCo	0.40 4%	0.52 0%	0.31 1%	0.27 0%	0.48 5%	0.21 1%	0.25 9%	0.42 0%	0.40 0%	0.36 2%
Eproxy	0.52 18%	0.06 10%	0.02 10%	0.29 15%	0.38 17%	0.31 13%	0.34 23%	0.31 11%	0.23 11%	0.27 14%
Eproxy + DPS	**0.57** 16%	**0.53** **35%**	0.30 18%	0.48 32%	**0.60** **24%**	**0.51** **13%**	0.39 **29%**	**0.65** **33%**	**0.59** **27%**	**0.51** **25%**
Cls-C Full training (∼4000GPU hrs)	0.29 **24%**	0.51 26%	**1.0** **100%**	**0.53** **34%**	0.21 16%	0.35 26%	0.17 14%	0.35 22%	0.37 25%	n/a N/A

Table 4.6: Comparison with predictor-based methods and efficient proxies on NAS-Bench-101. Eproxy+DPS as the fitness function for Regularization Evolutionary Algorithm can find near-optimal architectures with lower queries.

	RS	NAO	RE	Semi	WeakNAS			Eproxy+DPS		
Queries	2000	2000	2000	1000	200	150	100	150	60	0
Test Acc.	93.64	93.90	93.96	94.01	94.18	94.10	93.69	**94.23**	**93.92**	**93.07**

Table 4.7: Comparison with predictor-based methods on NAS-Bench-201 regarding the average queries required for retrieving the global optimal architectures. Eproxy+DPS uses substantially lower queries to find the global optimal architectures.

	Random Search	Regularized Evolution	MCTS	LaNAS	WeakNAS	Eproxy+DPS
C10	7782.1	563.2	528.3	247.1	182.1	**58.0 + 20**
C100	7621.2	438.2	405.4	187.5	78.4	$\mathbf{13.7_T}$
TinyImg	7726.1	715.1	578.2	292.4	268.4	$\mathbf{74.0_T}$

Table 4.8: Comparison with state-of-the-art NAS methods on ImageNet. $_{\mathrm{T}}$ stands for DPS is conducted in NDS search space and directly transferred to the target. Note Eproxy+DPS achieves the best results among NAS methods on CIFAR-10.

Method	Test Err. (%)		Params	FLOPS	Search Cost	Searched	Searched
	top-1	top-5	(M)	(M)	(GPU days)	Method	dataset
NASNet-A [29]	26.0	8.4	5.3	564	2000	RL	CIFAR-10
AmoebaNet-C [31]	24.3	7.6	6.4	570	3150	evolution	CIFAR-10
PNAS [104]	25.8	8.1	5.1	588	225	SMBO	CIFAR-10
DARTS(2nd order) [18]	26.7	8.7	4.7	574	4.0	gradient-based	CIFAR-10
SNAS [139]	27.3	9.2	4.3	522	1.5	gradient-based	CIFAR-10
GDAS [99]	26.0	8.5	5.3	581	0.21	gradient-based	CIFAR-10
P-DARTS [106]	24.4	7.4	4.9	557	0.3	gradient-based	CIFAR-10
P-DARTS	24.7	7.5	5.1	577	0.3	gradient-based	CIFAR-100
PC-DARTS [74]	25.1	7.8	5.3	586	0.1	gradient-based	CIFAR-10
TE-NAS [103]	26.2	8.3	6.3	-	0.05	training-free	CIFAR-10
PC-DARTS	24.2	7.3	5.3	597	3.8	gradient-based	ImageNet
ProxylessNAS [27]	24.9	7.5	7.1	465	8.3	gradient-based	ImageNet
TE-NAS [103]	24.5	7.5	5.4	599	0.17	training-free	ImageNet
Eproxy	25.7	8.1	4.9	542	0.02	evolution+proxy	CIFAR-10
Eproxy+DPS$_{\mathrm{T}}$	24.4	7.3	5.3	578	0.06	evolution+proxy	CIFAR-10

4.4 Summary

In this chapter, we proposed the Eproxy that utilizes a self-supervised few-shot regression task within near-zero cost. The Eproxy benefits from the barrier layer which significantly improves the complexity of the proxy task. To overcome the drawbacks of current efficient proxies that are not adaptive to various tasks/search spaces, we proposed DPS incorporating various settings and hyperparameters in a proxy search space and leveraging REA to conduct efficient exploration. Our experiments on numerous NAS benchmarks demonstrate that Eproxy is a robust, efficient proxy. Moreover, with the help of DPS, Eproxy achieves state-of-the-art results and outperforms existing state-of-the-art efficient proxies, early-stopping methods, and predictor-based NAS. Our work significantly ameliorates the inconsistency of efficient proxies and sets up a series of solid baselines while pointing out a novel direction for the NAS community.

CHAPTER 5

COMBINING HUMAN INSIGHT AND AUTOMATION VIA STRUCTURED GLOBAL CONVOLUTION

In the previous chapters, we discussed various approaches to improve the accuracy and efficiency of NAS. However, we also highlighted challenges in predicting the quality of neural architectures, particularly when dealing with large search spaces and diverse target tasks. Most NAS algorithms explore search spaces consisting of conventional components, such as small-kernel convolution layers, pooling layers, and hyperparameters like layer dimensions and downsampling rates. While these approaches have shown success, they often overlook the potential for novel architectural components that require human innovation, such as attention mechanisms [5].

In this section, we discuss how we can harness newly devised neural architecture components with automated search. We find that convolutional models have been widely adopted across various domains, but most rely on local convolution, which limits their ability to effectively capture long-range dependencies. Attention mechanisms, on the other hand, can aggregate global information using pairwise attention scores but come at the cost of increased computational complexity that grows quadratically with sequence length. This trade-off between capturing long-range dependencies and maintaining computational efficiency presents an opportunity for exploration. Inspired by the success of Gu et al.'s global convolutional model, S4, we propose a new approach to efficiently initialize the global convolutional kernel. By delving into the S4 model, we identify two key principles that contribute to its effectiveness: 1) efficient convolutional kernel parameterization that scales sub-linearly with sequence length, and 2) a decaying kernel structure where weights for convolving with closer neighbors are larger than those for more distant ones. Building upon these principles, we introduce Structured Global Convolution (SGConv), a simple yet powerful convolutional model that can efficiently capture long-range dependencies. To further optimize SGConv, we explore the sparsity of the kernel and utilize pruning to automatically

remove redundant weights. Furthermore, we search for a mixed structure in a search space with SGConv and attention blocks. By leveraging these techniques, we aim to create a more efficient and adaptable architecture that can be applied to a wide range of tasks. Our extensive experiments demonstrate the effectiveness of SGConv and its hybrid architecture. On the Long Range Arena and Speech Command datasets, our approach outperforms previous state-of-the-art methods while achieving faster training and inference speeds. Moreover, integrating SGConv into conventional language and vision models showcases its potential for improving both efficiency and quality.

SGConv highlights the importance of incorporating human insights into the design process. By understanding the key principles behind successful architectural designs, such as those in the S4 model, we can guide the search process toward more interpretable and efficient models. Our work paves the way for future research that combines human expertise with automated search techniques, enabling the discovery of robust and adaptable architectures for diverse tasks.

5.1 Introduction

Handling Long-Range Dependency (LRD) is a key challenge in long-sequence modeling tasks such as time-series forecasting, language modeling, and pixel-level image generation. Unfortunately, standard deep learning models fail to solve this problem for different reasons: Recurrent Neural Network (RNN) suffers from vanishing gradient, Transformer has complexity quadratic in the sequence length, and Convolutional Neural Network (CNN) usually only has a local receptive field in each layer.

A recently proposed benchmark called Long-Range Arena (LRA) [149] reveals that all existing models perform poorly in modeling LRD. Notably, all models fail on one spatial-level sequence modeling task called Pathfinder-X from LRA except a new Structured State Space sequence model (S4) [150]. The S4 model is inspired by the state space model widely used in control theory and can be computed efficiently with a special parameterization based on the Cauchy kernel. The exact implementation of the S4 model can be viewed as a *(depthwise) global convolutional model* with an involved computation global convolution kernel. Thanks to the global receptive field

of the convolution kernel, S4 can handle tasks that require LRD, such as Pathfinder [151, 149], where classic local CNNs fail [151, 152]. Also, the use of the Fast Fourier Transform (FFT) and techniques from numerical linear algebra make the computational complexity of S4 tractable compared to the quadratic complexity of attention. Together, S4 shows the potential of global convolutional models to model LRD and advance the SoTA on LRA.

Despite its accomplishments, the delicate design of the S4 makes it unfriendly even to knowledgeable researchers. In particular, the empirical success of S4 relies on 1) A Diagonal Plus Low Rank (DLPR) parameterization whose efficient implementation requires several numerical linear algebra tricks, 2) An initialization scheme based on the HiPPO matrix derived in prior work [153]. Therefore, aiming to reduce the complications of the model and highlight minimal principles, we raise the following questions:

What contributes to the success of the S4 model? Can we establish a simpler model based on minimal principles to handle long-range dependency?

To answer these questions, we focus on the design of the global convolution kernel. We extract two simple and intuitive principles that contribute to the success of the S4 kernel. The first principle is that the parameterization of the global convolution kernel should be efficient in terms of the sequence length: the number of parameters should scale slowly with the sequence length. For example, classic CNNs use a fixed kernel size. S4 also uses a fixed number of parameters to compute the convolution kernel while the number is greater than classic CNNs. Both models satisfy the first principle as the number of parameters does not scale with input length. The efficiency of parameterization is also necessary because the naive implementation of a global convolution kernel with the size of sentence length is intractable for inputs with thousands of tokens. Too many parameters will also cause overfitting, thus hurting the quality. The second principle is the decaying structure of the convolution kernel, meaning that the weights for convolving with closer neighbors are larger than the more distant ones. This structure appears ubiquitously in signal processing, with the well-known Gaussian filter as an example. The intuition is clear that closer neighbors provide a more helpful signal. S4 inherently enjoys this decaying property because of the exponential decay of the spectrum of matrix powers (See Figure 5.2), and we find this inductive bias improves the model quality (See Section 5.4.1).

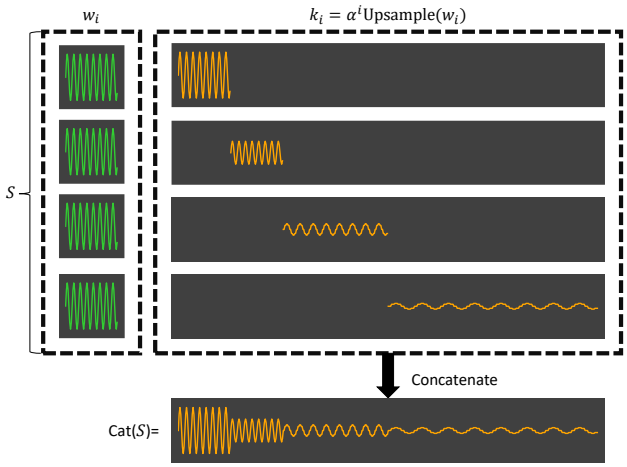


Figure 5.1: Illustration of the parameterization used in `SGConv` (Eq. (5.1)). The convolution kernel is composed of multi-scale sub-kernels.
Parameterization Efficiency. Every larger sub-kernel doubles the size of the previous sub-kernel while the same number of parameters are used for every scale, ensuring a logarithmic dependency of the number of parameters to the input length. **Decaying.** We use a weighted combination of sub-kernels where the weights are decaying, and smaller weights are assigned to larger scales.

We show that these two principles are sufficient for designing a global convolutional model that captures LRD well. To verify this, we introduce a class of global convolution kernels with a simple *multiscale* structure, as shown in Figure 5.1. Specifically, we compose the convolution kernel by a sequence of sub-kernels of increasing sizes, yet every sub-kernel is upsampled from the same number of parameters. This parameterization ensures that the number of parameters only scales logarithmically to the input length, which satisfies the first principle. In addition, we add a decaying weight to each scale during the combination step and fulfill the second principle. We named our methods as Structural Global Convolution kernels (`SGConv`). Empirically, `SGConv` improves S4 by more than 1% and achieves SoTA results on the LRA benchmark. On Speech Command datasets, `SGConv` achieves comparative results in the ten-class classification task and significantly better results in the 35-class classification task upon previous SoTA. We further show that `SGConv` is more efficient than S4 and can be used as a general-purpose module in different domains. For example, a hybrid model of classic attention and `SGConv` shows promising quality on both autoregressive language modeling and sentence classification tasks, replacing the 2D convolution kernel of the

ConvNext model with 1D `SGConv` matches the quality of the original model. Moreover, we propose an automatic pruning strategy to prune the kernel efficiently in the post-training phase.

From the perspective of NAS, the `SGConv` can be interpreted as a kernel-level fine-grained search for the distribution of parameters by utilizing parameterization. Furthermore, the `SGConv` has shown that the global convolution kernel exhibits sparsity and can be pruned (Figure 5.9), meaning that the effective kernel length can be automatically determined through the training phase. These findings can potentially spark further research and development in the field. Another simple approach we explore in NAS is the combination of Attention and `SGConv` through a mixture model (Section 5.4.3). This approach is both intuitive and efficient and has the potential to improve the quality of neural network architectures further.

5.2 Related work

Efficient Transformers. The Transformer architecture [5] has been successful across a wide range of applications [3, 154, 155, 156] in machine learning. However, the computation and memory complexity of the Transformer scales quadratically with the input length, making it intractable for modeling long-range interactions in very long sequences. Therefore, several efficient variants of the Transformer model have been proposed recently to overcome this issue [157, 21, 158, 159, 160, 161, 162]. Nevertheless, few of these methods performed well on benchmarks such as Long Range Arena [149], and SCROLLS [163], which require long-range modeling ability.

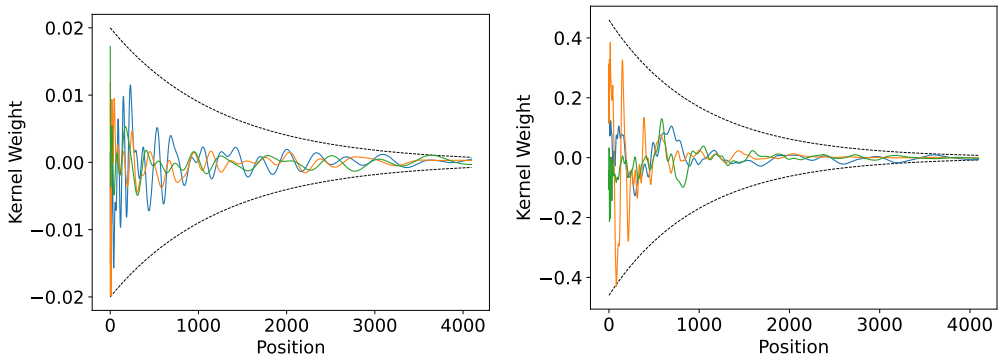
(Re-)parameterization. Parameterization is a crucial but underrated part of architecture design because different parameterizations usually provide different inductive biases. For example, weight normalization [164] parameterizes the norm and direction of the weight matrices separately and thus reaches faster convergence. On the other hand, Zagoruyko et al. [165] proposed a Dirac weight re-parameterization to train deep networks without explicit skip-connections and matched the quality of ResNets [62]. In computer vision, several works explored using structural re-parameterization to create 2D convolution kernels. Most of these works [166, 167, 168, 169] are limited

to the vision domain and utilize only short-range convolution kernels (e.g., 7×7) except for the line of work based on 2D Fourier operators [170, 171] and the line of work based on continuous convolutional kernel [172, 173, 174]. Our `SGConv` kernel is a special parameterization of global convolution kernels that tackles LRD and showcases the extensibility of re-parameterized kernels.

State Space Models. The state space model (SSM) uses a set of linear differential equations to model physical systems with input, output, and state variables. It is widely used in control, neuroscience, and statistics. Recently, Gu et al. [175] introduced a deep SSM-based model termed S4 that can outperform prior approaches on several long sequence modeling tasks with a specially structured state transition matrix. However, the expensive computation and memory requirements make it impractical. A follow-up work of S4 proposed a new parameterization of SSM [150], which decomposes the state transition matrix into the sum of low-rank and normal matrices and implements SSM as a global convolutional model. Under this parameterization, the authors then combine the techniques of diagonalizing the Cauchy kernel and performing low-rank corrections with the Woodbury identity to compute the global convolution kernel. While achieving promising results, S4 is theoretically involved and practical implementations of S4 require accelerator-specific dedicated code optimization for the Cauchy kernel computation. This makes it difficult to readily implement in deep learning frameworks [176, 177, 178, 179] and hardware targets. Concurrent with this work, many state-space-based models are emerging and bringing better quality [180, 181, 182].

5.3 Design of Global Convolutional Models

We summarize the design principles that enable the global convolutional model to be both efficient and effective. Then we introduce the proposed Structured Global Convolution (`SGConv`) based on the highlighted principles.



(a) Pathfinder-X (b) SC-10

Figure 5.2: Visualization of S4 kernels on (a) Pathfinder-X and (b) Speech Command 10-class. The values in the convolution kernel exhibit a decaying behavior. We only plot the first 4096 positions for better illustration.

5.3.1 Design Principles

The two intuitive design principles that contribute to the success of S4 are efficient parameterization and decaying structure.

Efficient Parameterization. Different from local convolution, where the kernel size is fixed, global convolution requires a kernel size that is the same as the sentence length. Naive parameterization of convolution kernel as classic local convolutions is therefore intractable for long sequences. For instance, the Pathfinder-X task has a length of $16K$. It then impractically requires $4M$ parameters for a single layer to model the depth-wise global convolution kernel with a standard channel size of 256. Thus, an efficient convolution kernel parameterization is necessary, especially when the sentence is extremely long. For example, S4 takes a well-designed Normal Plus Low-Rank (NPLR) parameterization to model the whole kernel with two special matrices where the number of parameters is fixed.

Decaying Structure. Apart from the efficiency of the parameterization, we find that a decaying structure of the convolution kernel provides a good inductive bias to long-sequence modeling and contributes to the quality (See Section 5.4.1 for detailed ablation study). Concretely, the magnitude of the value in the convolution kernel should decay so that more weight is assigned to the close neighbors. S4 model inherently satisfies this property because

the k-th element of the kernel of S4 is $\mathbf{C}\mathbf{A}^k\mathbf{B}$ and the operator norm of the power of a matrix decays exponentially:

Fact 1. *For a square matrix* $\mathbf{A}$*, the operator norm* $\left\|\mathbf{A}^k\right\|_2 \leq \|\mathbf{A}\|_2^k$*. In particular, if* $\|\mathbf{A}\|_2 < 1$*,* $\left\|\mathbf{A}^k\right\|_2$ *decays exponential to* k*, so* $\left\|\mathbf{C}\mathbf{A}^k\mathbf{B}\right\|_2 \leq \|\mathbf{C}\|_2 \left\|\mathbf{A}^k\right\|_2 \|\mathbf{B}\|_2$ *also decays exponentially.*

We can also directly observe the decaying structure of S4 in different tasks in Figure 5.2.

5.3.2 SGConv

Putting the two principles together, we propose a simple global depth-wise convolution, dubbed Structured Global Convolution (`SGConv`), based on multiscale sub-kernels and weighted combinations. (See Figure 5.1). We will first introduce the parameterization of the convolutional kernel and then present how to build a global convolutional model with this kernel.

Parameterization of SGConv Kernel. Formally, let L be the length of the input sequence, the convolutional kernel should also has length L. We define the parameter set of a single channel as $S = \left\{\mathbf{w}_i | 0 \leq i < \left[\log_2\left(\frac{L}{d}\right)\right] + 1\right\}$ where $\mathbf{w}_i \in \mathbb{R}^d$ is the parameter for i-th sub-kernel k_i, and d is the dimension of the parameter. Denote the number of scales $N = \left[\log_2\left(\frac{L}{d}\right)\right] + 1$. We use the upsample operation, implemented as linear interpolation, to form sub-kernels of different scales. We use $\text{Upsample}_l(\mathbf{x})$ to denote upsampling $\mathbf{x}$ to length l (We use `F.interpolate` function in Pytorch and set the mode to be `linear` in our implementation). We also introduce a normalization constant Z to ensure the convolution operation will not change the scale of the input and a coefficient α to control the decaying speed. Now, we are ready to introduce the weighted combination scheme by concatenating a set of weighted sub-kernels k_i:

$$\text{Cat}(S) = \frac{1}{Z}\left[k_0, k_1, \cdots, k_{N-1}\right], \text{ where } k_i = \alpha^i \text{Upsample}_{2^{\max[i-1,0]}d}\left(\mathbf{w}_i\right). \tag{5.1}$$

It is easy to check that $\text{Cat}(S)$ gives the convolution kernel with length $\sum_{i=0}^{N} 2^{\max[i-1,0]}d = 2^{N-1}d \geq L$ (See Figure 5.1 for an illustration), which

can be truncated to L if it is overlength. And the number of parameters is $Nd = O(\log L)$. The decay coefficient α, usually chosen to be $1/2$, induces the decaying structure.

Incorporate SGConv to Modern Architectures. In the implementation, we compute the depth-wise convolution kernel and use Fast Fourier Transform to compute the convolution in $O(L \log L)$ time (See Figure 5.3 for detailed illustration). We compute the normalization constant Z such that the norm of the kernel is one at initialization and fix it during training. Please refer to Listing 5.1 for a Python-style pseudo-code. We can plug `SGConv` into modern architectures as a replacement of attention in Transformer or local convolution in ConvNets (See Figure 5.5, 5.8 for two examples). Due to the relaxation of the structure of the convolutional kernel, `SGConv` does not have the RNN-style reformulation as S4. Yet, `SGConv` is naturally capable of performing autoregressive generation, such as language modeling, similarly to classic causal convolutional models [183, 184] and Transformers. Concretely, the convolution kernel is unidirectional, where the computation at the embedding of i-th is only computed based on tokens before i, and left zero padding is used for ignoring the overlength kernel. During generation, hidden states of past tokens are cached for fast calculation of the next token with a single convolution step. Due to the simplicity of the parameterization, `SGConv` kernel is easy to compute and more efficient than the S4 kernel, as shown in Section 5.4.1.

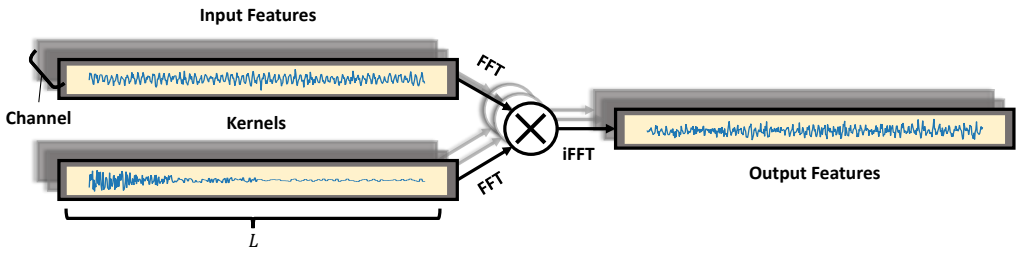


Figure 5.3: Implementing `SGConv` with FFT. We first compute the convolutional kernels for each channel as described in Section 5.3.2, and apply the depth-wise global convolution to the input features.

```
# Parameters
kernel_param_list = [] # w_i
```

```
for _ in range(num_scales):
    kernel_param_list.append(
        nn.Parameter(torch.randn(hidden_dim, kernel_dim))
    ) # size: h * d

# Compute global convolution kernel
kernel_list = [] # k_i
for i in range(num_scales):
    kernel = F.interpolate(
        kernel_param_list[i],
        scale_factor = 2**max(0, i-1),
        mode = "linear"
    ) * 0.5 ** i # alpha = 0.5
    kernel_list.append(kernel)
# The computed kernel, size: h * (d * 2^{s-1})
k = torch.cat(kernel_list, dim=-1)

# Normalize kernel
if is_init: # Compute the norm at initialization
    kernel_norm = k.norm(dim=-1, keepdim=True).detach()
k = k / kernel_norm

# Use kernel to compute global convolution
# x: batch_size * hidden_dim * seq_len
L = x.size(-1)
# Truncate kernel if it is too long
k = k[..., :L]
# Use FFT to compute convolution
x_f = torch.fft.rfft(x, n=2*L)
k_f = torch.fft.rfft(k, n=2*L)
y_f = torch.einsum("b h l, h l -> b h l", x_f, k_f)

# Inverse FFT to get the result
y = torch.fft.irfft(y_f, n=2*L)[..., :L]
```

Listing 5.1: Pseudo PyTorch-sytle code for SGConv.

5.4 Experiments

In this section, we first test the effectiveness of `SGConv` on two standard long sequence modeling tasks, i.e., Long Range Arena [149] and Speech Com-

Table 5.1: Hyperparameters used in LRA experiments.

	ListOps	Text	Retrieval	Image	Pathfinder	Path-X
Acc.	61.45	89.20	91.11	87.97	95.46	97.83
Scale dim.	1	2	1	32	32	64
Dropout	0	0	0	0.2	0.2	0

Table 5.2: The quality of `SGConv` compared to other baselines on the LRA dataset. `SGConv` achieves significant improvement compared to previous methods with a more straightforward structure and faster speed (See Table 5.3)

Model	ListOps	Text	Retrieval	Image	Pathfinder	Path-X	Avg.
Transformer	36.37	64.27	57.46	42.44	71.40	✗	54.39
Sparse Trans.	17.07	63.58	59.59	44.24	71.71	✗	51.24
Linformer	35.70	53.94	52.27	38.56	76.34	✗	51.36
Reformer	37.27	56.10	53.40	38.07	68.50	✗	50.67
BigBird	36.05	64.02	59.29	40.83	74.87	✗	55.01
S4 (original)	58.35	76.02	87.09	87.26	86.05	88.10	80.48
S4 [185]	59.60	86.82	**90.90**	**88.65**	94.20	96.35	86.09
SGConv	**61.45**	**89.20**	**91.11**	87.97	**95.46**	**97.83**	**87.17**

mands [186], and compare it with S4 and other baselines. We also conduct ablation studies over the decay speed and scale dimension d and evaluate the speed of `SGConv` on LRA. Further, we explore the possibility of plugging the global convolutional layer into standard models as a *general-purpose component* for capturing long-range dependency. For language tasks, we find that replacing half of layers of Transformer with a certain strategy with `SGConv` block will not hurt quality, while the complexity of those layers improves from $O(L^2)$ to $O(L \log L)$. On ImageNet, we replace the 7×7 convolution in ConvNext [187] with `SGConv` and show comparative or better quality.

5.4.1 Long Range Arena

Long Range Arena benchmark [149] is a suite of six tasks consisting of sequences ranging from 1K to 16K tokens, encompassing a wide range of data types and modalities such as text, natural, synthetic images, and mathematical expressions requiring similarity, structural, and visual-spatial reasoning.

Table 5.3: Comparison of the inference and backpropagation time (ms/batch) of S4 and `SGConv` blocks (number of channels 128, batch size 64) on CPU and GPU. Note that the parameterization in S4 requires a customized CUDA kernel to improve the efficiency (refer to opt. in the Table). Nevertheless, `SGConv` still *always* surpasses S4 even compared to the optimized CUDA kernel.

	Sequence length	256	512	1024	2048	4096	8192	16384
Inf. CPU	S4	29.4	81.7	158.3	306.9	594	1156.9	2274.0
	SGConv	**23.8**	**56.2**	**108.7**	**211.3**	**409.3**	**789.5**	**1559.3**
Inf. GPU	S4 (w/o opt)	2.7	2.7	4.4	7.9	15.2	32.7	64.5
	S4 (w. opt.)	1.6	1.9	3.1	5.4	10.0	22.3	44.3
	SGConv	**1.2**	**1.3**	**2.3**	**4.4**	**8.5**	**19.8**	**39.4**
BP GPU	S4 (w/o opt)	4.1	5.7	10.2	19.4	38.1	80.1	161.2
	S4 (w. opt.)	3.5	4	6.6	11.9	22.6	48.9	97.8
	SGConv	**2.0**	**2.7**	**5.0**	**9.6**	**18.6**	**41.2**	**82.5**

5.4.1.1 Results

We show the experimental results in Table 5.2 with several baseline methods [5, 157, 21, 158, 159, 150, 185]. `SGConv` achieves a 1% improvement in average accuracy upon well-tuned S4 variants introduced in the work [185]. Notably, `SGConv` is guided by the two intuitive principles and has a much simpler structure than S4 [185]. The detailed implementation settings can be found in Table 5.1.

5.4.1.2 Ablation Study on IMDB

We conduct ablation studies on the IMDB byte-level document classification task in the LRA benchmark. We mainly focus on two aspects: 1) The speed of decaying and 2) The parameter dimension d of each scale. For simplicity, in the standard `SGConv` formulation (Eq. (5.1)), we fix the decay coefficient $\alpha = 1/2$ and only tune the dimension d. However, the actual decay speed as a function of the position in the kernel depends both on α and d, making it hard to conduct ablation studies. Thus, we use a slightly different convolution kernel that disentangles the decay speed and the dimension of each scale:

$$\begin{aligned} \mathrm{Cat}^*(S) = \frac{1}{Z}\left[k_0, k_1, \cdots, k_{N-1}\right] \odot \left[\frac{1}{1^t}, \frac{1}{2^t}, \cdots, \frac{1}{L^t}\right], \\ \text{where } k_i = \mathrm{Upsample}_{2^{\max[i-1,0]}d}\left(\mathbf{w}_i\right). \end{aligned} \tag{5.2}$$

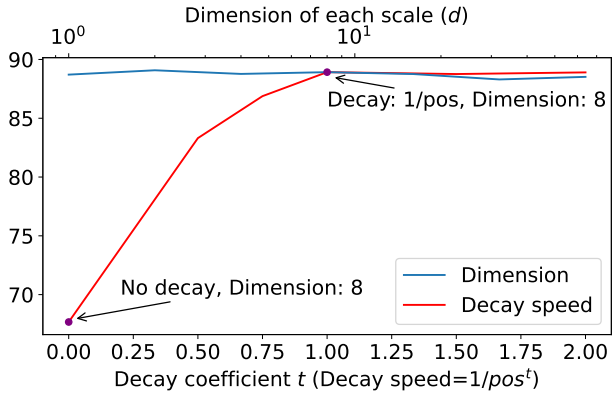


Figure 5.4: Ablation study on the effect of decay speed and hidden dimension of each scale on IMDB dataset. $pos \in [1, L]$ refers to the position in the convolution kernel. We observe: 1) The decay structure is crucial for getting good quality; 2) In a reasonable range, d (Dimension) has less impact on the quality than t ($t \in [0, 2.0]$).

t here then controls the decay speed, which is independent of each scale's dimension. We conduct two sets of experiments: 1) Fix $d = 8$, vary t from 0 (which means no decay) to 2, and 2) Fix $t = 1$, vary d from 1 to 64. Figure 5.4 reports the accuracies in different settings. We can observe that 1) The decay structure is crucial for getting good quality, and 2) In a reasonable range, d has less impact on the quality than t. Nevertheless, we observe a trend of quality drop when increasing d from 8 to 64. Experiments on larger d show worse quality, which can be attributed to overfitting.

5.4.1.3 Speed Comparison

In Table 5.3, we compare the computation speed of the S4 kernel and `SGConv` kernel in different settings. Due to its simplicity, `SGConv` is faster than S4 for any sentence length. `SGConv` is about 50% faster than the vanilla implementation of the S4 kernel and is 15% faster than the optimized CUDA kernel implementation without resorting to optimized CUDA kernels.

5.4.2 Speech Commands

The Speech Command (SC) dataset [186] is a 35-class dataset of one second (16000 HZ sampling rate) spoken words in English. However, followup

works [188, 175, 172, 173] adopted a smaller 10-class subset of SC. And works [173, 175] on the SC dataset specifically use pre-processing such as MFCC features. Our baselines are obtained from the works [150, 180]. Note that besides SSM-based models, there is no strong baseline for raw waveform classification using either the 10-class or the full dataset. And SSM-based methods also show the ability to perform 0-shot testing at lower sampling rate such as 8000 Hz. Table 5.4 shows that the `SGConv` yields better results compared to the SSM-based method among four out of five tasks. Notably, for the original SC (35-class), `SGConv` achieves marginally higher accuracy for raw-sequence classification and significantly better results (+2.40%) compared to the existing SoTA method.

Table 5.4: Speech Command classification results compared to existing methods. * We carefully reproduce the S4 method based on the released code[1]. Since the latest version removed 10-class experiments settings, we utilized a earlier version[2].The results suggest that for the SC 35-classification, `SGConv` achieves SoTA on both full length task and 2X sampling rate, zero-shot task.

10-cls	Transformer	Performer	NRDE	CKConv	WaveGAN-D	S4	S4*	SGConv
MFCC	90.75	80.85	89.8	**95.3**	✗	93.96	92.05	94.91
16000HZ	✗	30.77	16.49	11.6	71.66	98.32	**97.98**	97.52
8000HZ (0-shot)	✗	30.68	15.12	65.96	✗	96.30	91.83	**96.03**

35-cls	InceptionNet	ResNet-18	XResNet-50	ConvNet	S4D	S4	S4*	SGConv
16000HZ	61.24	77.86	83.01	95.51	96.25	96.08	96.27	**96.42**
8000HZ (0-shot)	5.18	8.74	7.72	7.26	91.58	91.32	91.89	**94.29**

5.4.3 Further Applications of SGConv

We further study `SGConv` as a generic network architecture *drop-in* component targeting tasks in language modeling and computer vision. First, we present an efficient mixture of attention and `SGConv` layers architecture that replaces half of the attention blocks in the Transformer with the `SGConv` blocks. We demonstrate the potential of utilizing such a model for long text processing. Then we incorporate `SGConv` (1D) into ConvNeXt [187]. Surprisingly, `SGConv` achieves comparable or even better results compared to several SoTA CNN and Vision Transformer models by treating the 2D features as

[1] `https://github.com/HazyResearch/state-spaces`

[2] `https://github.com/HazyResearch/state-spaces/tree/307f11bba801d5734235a1791df1859f6ae0e367`

a 1D sequence. Furthermore, we deliver an efficient post-training automatic pruning strategy for the proposed neural architecture.

Table 5.5: quality comparison on WikiText-103.

Model	Valid.	Test
LSTM+Hebb.	29.0	29.2
16L Transformer-XL	-	24.0
16L SGConv+SAttn	21.90	**22.83**
Adaptive Input	-	18.7
S4	-	20.95
18L Transformer-XL	-	**18.3**
18L Transformer-XL*	18.16	18.75
18L SGConv+SAttn	18.10	18.70

Table 5.6: Comparison of inference time and GPU memory utilization with Attention blocks. `SGConv` has significantly less memory usage and faster inference speed when the sequence increases.

		256	512	1024	2048	3072
Attn. Block	Inf. (ms/batch)	**2.6**	7.3	23.2	91.7	✗
	Mem. (GB)	**2.6**	3.9	7.9	23.9	OOM
SGConv Block	Inf. (ms/batch)	2.7	**5.4**	**10.9**	**21.8**	**43.6**
	Mem. (GB)	**2.6**	**3.4**	**5.2**	**8.7**	**15.7**

Language modeling. We propose the `SGConv` block (shown in Figure 5.5) which is similar to the Attention block in Transformer [5]. `SGConv` block enjoys both $O(L\log(L))$ time complexity and space complexity. We benchmark the inference time and GPU memory usage of both `SGConv` and Attention in Table 5.6. When the sequence length is 1024, `SGConv` block is $\sim$2.1X faster than the Attention block. For language modeling, we use the feature of `SGConv` to process the long sequences directly. The Attention block only targets the short-range data termed SAttention. We illustrate the structure in Figure 5.6a. Furthermore, we investigate the strategy to replace the Attention blocks with `SGConv` blocks. To perform the Neural Architecture Search Analysis, we generate 50 architectures with eight `SGConv` blocks and eight

Attention blocks where the order is shuffled. We denote the average depth to replace the Attention blocks as $\sum_{i=0}^{N_{SGConv}} \text{idx}_i / N_{total}$ where the idx denotes the ith `SGConv` depth position. $N_{SGConv} = 8$ and $N_{total} = 16$ in this case. The results in Figure 5.6b strongly suggest that when fixing the number of `SGConv` layer, models achieve better quality by placing `SGConv` blocks in *deeper* layers. Guided by the strategy, we handcraft two Transformer-XL [189] style models. (1) 16-layer: {A, A, A, C}×2 + {A, C, C, C}×2. (2) 18-layer: {A, A, C}×3 + {A, C, C}×3. A denotes SAttention and C denotes `SGConv`. $\times N$ denotes repeating the order of layers for N times. We test the model on WikiText-103 [190] which is a wide-used language modeling benchmark with an average length of 3.6K tokens per article. We set both the attention and memory length to 384 for 18L model and 192 for 16L model. The length of input sequence is 3092 which can be processed by `SGConv` directly. We show the results in Table 5.5. Our results suggest that when the attention range is short, the 16L model outperform the baseline with -1.17 perplexity. For the 18L model, our model achieves 18.70 perplexity. Note that we use a smaller and affordable batch size (16) for training. Under the same setting, our model gains slightly better perplexity than Transformer-XL (-0.05). Our results show the potential of adopting `SGConv` as part of the language model for long range language sequence processing.

Sentence classification. We combine the `SGConv` block with the BERT model [7]. Concretely, we utilize the 12-layer {A, A, C}×2+{A, C, C}×2 model. The pretraining is conducted on BooksCorpus [191] and English Wikipedia [192]. We then fine-tune the model on the GLUE benchmark [193]. To avoid the instability of fine-tuning on small datasets, we only test on tasks with more than $5K$ training samples. We follow the training and fine-tuning pipeline of [194] (BERT-A in Table 1 of [194]) and report the average accuracy of five different random seeds. `SGConvBERT` achieves comparable quality to the original BERT model, while the `SGConv` layer is more efficient than the attention layer.

Image Classification We also evaluate the adaptability of `SGConv` by applying it to large-scale image classification. We conduct experiments on ImageNet-1k [102] which consists of more than 1.28 million high-resolution training and 50,000 validation images. We use the training settings in the

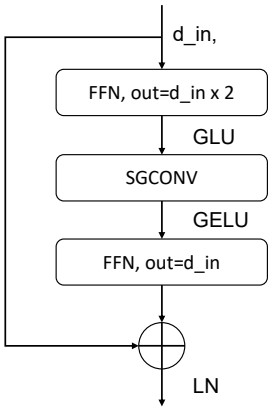
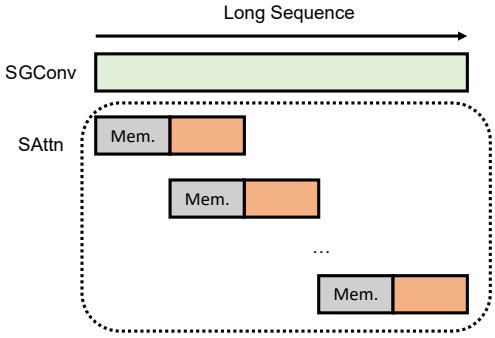
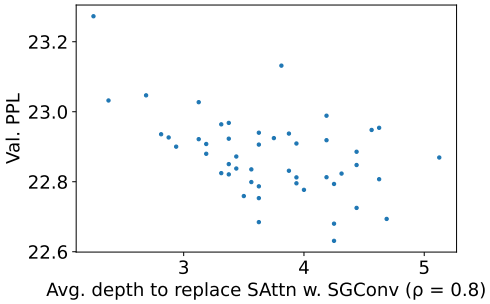


Figure 5.5: `SGConv` block



(a) Illustration of `SGConv` and Transformer-XL style Short Attention used in language modeling task. `SGConv` directly processes the full length sequence.



(b) The depth to replace SAttention with `SGConv` vs. validation PPL on WikiText-103

Figure 5.6: Incorporating `SGConv` to Transformer models in language tasks.

work [187][3]. We replace the 7×7 2D convolutional kernels with `SGConvs` in ConvNeXt [187] denoted as `SGConvNeXt`. The block designs of `SGConvNeXt` are shown in Figure 5.8. Note we train various sizes of `SGConveNeXt` using hyperparameter settings from ConvNeXt[3] without any changes. By treating the 2D features as sequences, our `SGConvNeXt` achieves better results compared to existing SoTA methods such as EfficientNets [195], Swin Transformers [154] (shown in Figure 5.7). Note that Vision Transformer [3] and its variants [196, 197, 198] adopt patching techniques that can lead to a quadratic increase in complexity with image size. Also, patching is incompatible with

[3] `https://github.com/facebookresearch/ConvNeXt`

Table 5.7: quality comparison of BERT and `SGConvBERT` on GLUE dataset. `SGConvBERT` is comparable with BERT while being more efficient. We exclude MRPC and RTE datasets in GLUE because their sizes are too small ($< 5K$ training samples).

	MNLI-m/mm	QNLI	QQP	SST	CoLA	STS	Avg.
BERT	**84.93/84.91**	**91.34**	91.04	**92.88**	55.19	88.29	84.08
SGConvBERT	84.78/84.70	91.25	**91.18**	92.55	**57.92**	**88.42**	**84.40**

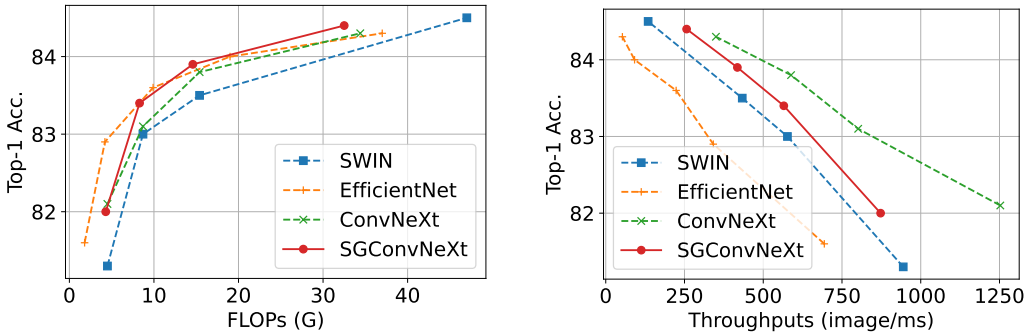


Figure 5.7: Comparison of ImageNet-1k Top-1 accuracy with SoTA works. Left: Top-1 Accuracy vs. FLOPs. Right: Top-1 Accuracy vs. Throughputs.

dynamic input resolutions while `SGConvNeXt` processes the data globally. We list several interesting directions that can be explored for future work: 1) Optimization for the long-range convolution: we noticed that though FFT theoretically requires less FLOPs than plain convolution, the throughput drops empirically. One reason is that there is no optimized CUDA implementation for 1D long-range convolution and can be a good direction for future work. 2) Optimized hyperparameters and data augmentation methods: ConvNeXts' hyperparameters are tuned for maximum quality, which may not be ideal for `SGConvNeXt`. 3) `SGConv` for vision reasoning tasks: we show that `SGConv` is powerful for long-range synthetic reasoning tasks and large-scale classification tasks. It could be effective in visual reasoning applications such as Vision-Language Reasoning [199, 200] with great potential.

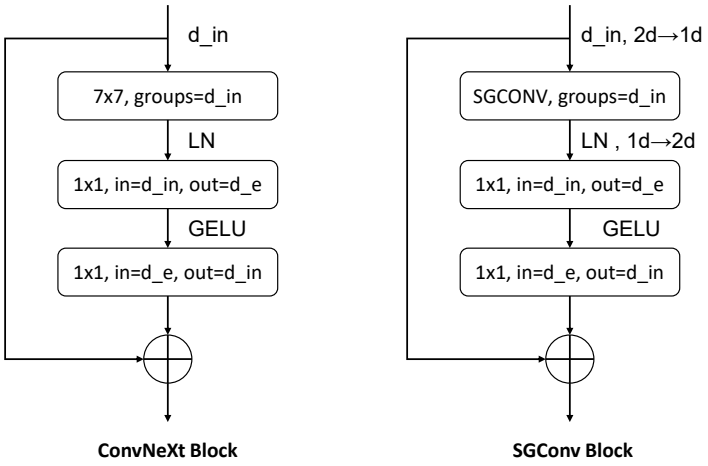
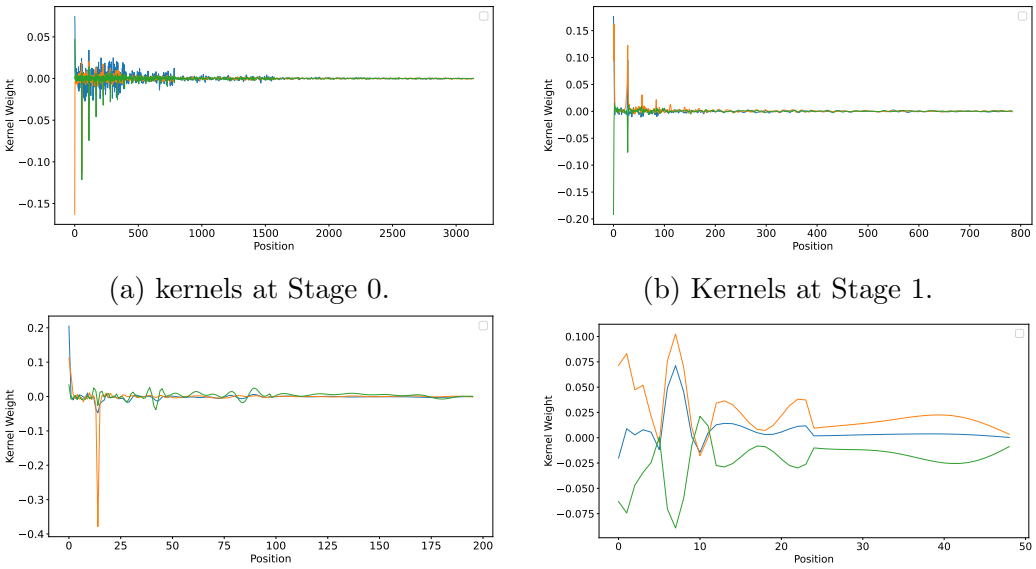


Figure 5.8: `SGConvnext`

NAS-driven Kernel Pruning To explore the impact of pruning on the trained `SGConv` kernel and highlight the kernel-level fine-grained search for parameter distribution through re-parameterization as NAS, we carried out a series of experiments using a `ConvNext`-Tiny. We observe the decay for post-training kernels (Shown in Figure 5.9). Thus, we implement pruning on the `SGConv` kernel by modifying the pruning ratio from 0.1 to 0.9 in increments of 0.1. The pruning targeted the kernel of the trained `SGConv` layer. We test the model with the pruned kernel and without training and assessed the test set accuracy. As illustrated in Figure 5.10, accuracy declined as the pruning ratio increased. However, even at a pruning ratio of 0.9, the accuracy only dropped to 75.43%, remaining notably higher than the majority of efficient CNNs. This implies that the `SGConv` kernel can maintain its effectiveness despite a substantial degree of sparsity. These findings reveal that pruning can be employed on the `SGConv` kernel without considerable accuracy loss, potentially paving the way for more efficient and resource-conscious models in the future, through the application of NAS-based parameterization.



(a) kernels at Stage 0.

(b) Kernels at Stage 1.

(c) Kernels at Stage 2.

(d) Kernels at Stage 3.

Figure 5.9: Kernels in `SGConvNeXt` at different stages.

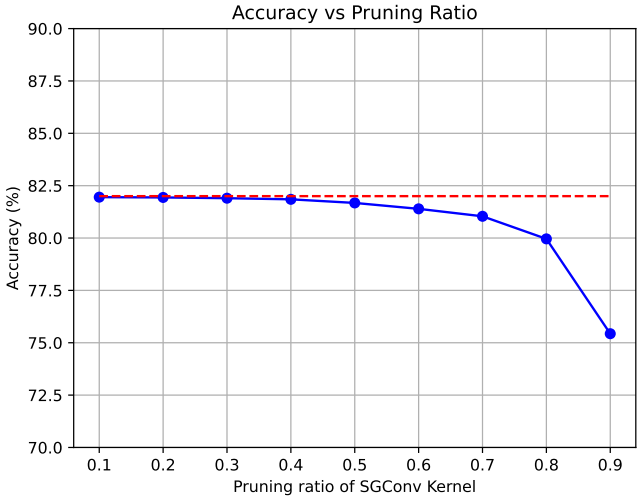


Figure 5.10: Kernel pruning on `SGConvNext`-Tiny.

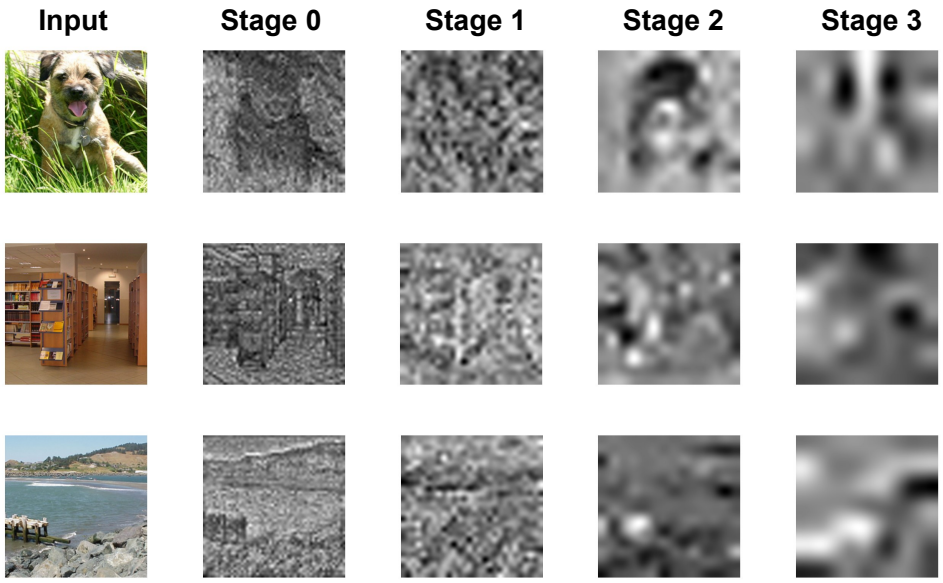


Figure 5.11: Visualization of the intermediate features of `SGConvNeXt` on ImageNet-1k dataset.

5.5 Summary

In this chapter, we delve into the factors contributing to the success of convolutional models in long-sequence modeling tasks and identify two key principles. Based on these principles, we propose a simple and intuitive global convolutional model, `SGConv`, demonstrating both direct implications and robust quality. Simultaneously, other works simplify the S4 model by constraining the state transition matrix to be diagonal [180, 201]. The work[180] introduces a sophisticated approach to parameterization and initialization schemes in contrast to our work. Their method offers insights into the S4 phenomenon from a state-space-model perspective. As long-range dependency becomes increasingly crucial for sequence modeling, we believe that similar global convolutional modules will continue to emerge in the future. From a neural architecture search (NAS) perspective, our work on `SGConv` offers an innovative approach to understanding and improving the quality of neural network architectures, opening new avenues for research and development in the field.

CHAPTER 6

AUTOMATED DESIGN OF DECODING ADAPTER TOWARDS LARGE LANGUAGE MODELS

In previous chapters, we explored various methods to efficiently and accurately automate neural architecture design. Furthermore, we examined how to leverage automation to optimize innovative neural components. This chapter focuses on addressing some limitations of current Large Language Models (LLMs) that adhere to the same paradigm. Inspired by keen observations and innovative designs, we investigate how automation can be employed to explore the design space of both software and hardware for LLMs.

First, we observe that Large Language Models employ auto-regressive decoding, which requires sequential computation with each step reliant on the output of the previous one. This creates a bottleneck, as each step necessitates moving the full model parameters from High-Bandwidth Memory (HBM) to the accelerator's cache.

While methods such as speculative decoding have been suggested to address this issue, their implementation is impeded by challenges associated with acquiring and maintaining a separate draft model.

We present Medusa, an efficient method that augments LLM inference by adding extra decoding heads to predict multiple subsequent tokens in parallel. Using a tree-based attention mechanism, Medusa constructs multiple candidate continuations and verifies them simultaneously in each decoding step. By leveraging parallel processing, Medusa reduces the number of decoding steps required.

We present two levels of fine-tuning procedures for Medusa to meet the needs of different use cases: Medusa-1, where Medusa is directly fine-tuned on top of a frozen backbone LLM, enabling lossless inference acceleration; and Medusa-2, where Medusa is fine-tuned together with the backbone LLM, enhancing the prediction accuracy of Medusa heads and achieving higher speedup, but requiring a special training recipe to preserve the model's capabilities.

Moreover, we propose several automated strategies that improve or expand the utility of Medusa, including a *searched tree attention* to explore the sparsity of tree attention and search for the optimal number of decoding heads, a *self-distillation* to automatically distilled from the model where no training data is available, and a *typical acceptance scheme* to boost the acceptance rate while maintaining generation quality.

We evaluate Medusa on models of various sizes and training procedures. Our experiments demonstrate that Medusa-1 can achieve over 2.2× speedup without compromising generation quality, while Medusa-2 further improves the speedup to 2.3-2.8×.

Finally, we investigate hardware constraints and introduce an analytical model to predict Medusa's performance across various model sizes, sequence lengths, and batch sizes on target devices. This approach enables efficient analysis of Medusa's scalability to larger models on new hardware.

6.1 Introduction

The recent advancements in Large Language Models (LLMs) have demonstrated that the quality of language generation significantly improves with an increase in model size, reaching billions of parameters [8, 202, 203, 204, 205, 206, 20]. However, this growth has led to an increase in *inference latency*, which poses a significant challenge in practical applications. From a system perspective, LLM inference is predominantly memory-bandwidth-bound [207, 208], with the main latency bottleneck stemming from accelerators' memory bandwidth rather than arithmetic computations. This bottleneck is inherent to the sequential nature of auto-regressive decoding, where each forward pass requires transferring the complete model parameters from High-Bandwidth Memory (HBM) to the accelerator's cache. This process, which generates only a single token, underutilizes the arithmetic computation potential of modern accelerators, leading to inefficiency.

To address this, one approach to speed up LLM inference involves *increasing the operational intensity* (the ratio of total floating-point operations (FLOPs) to total data movement) of the decoding process and *reducing the number of decoding steps*. In line with this idea, speculative decoding has been proposed [209, 210, 211, 212]. This method uses a smaller draft model

to generate a token sequence, which is then refined by the original, larger model for acceptable continuation. However, obtaining an appropriate draft model remains challenging, and it's even harder to integrate the draft model into a distributed system [210].

Instead of using a separate draft model to sequentially generate candidate outputs, in this chapter, we revisit and refine the concept of using multiple decoding heads on top of the backbone model to expedite inference [213]. We find that when applied effectively, this technique can overcome the challenges of speculative decoding, allowing for seamless integration into existing LLM systems. Specifically, we introduce Medusa, a method that enhances LLM inference by integrating additional decoding heads to concurrently predict multiple tokens. These heads are fine-tuned in a *parameter-efficient* manner and can be added to any existing model. With no requirement for a draft model, Medusa offers easy integration into current LLM systems, including those in distributed environments, ensuring a user-friendly experience.

We further enhance Medusa with two key insights. Firstly, the current approach of generating a single candidate continuation at each decoding step leads to inefficient use of computational resources. To address this, we propose generating multiple candidate continuations using the Medusa heads and verifying them concurrently through a simple adjustment to the attention mask. Secondly, we can reuse the rejection sampling scheme as used in speculative decoding [209, 210] to generate consistent responses with the same distribution as the original model. However, it cannot further enhance the acceleration rate. Alternatively, we also introduce a *typical acceptance* scheme that selects *reasonable* candidates from the Medusa head outputs. We use temperature as a threshold to manage deviation from the original model's predictions, providing an efficient alternative to the rejection sampling method.

To equip LLMs with predictive Medusa heads, we propose two distinct fine-tuning procedures tailored to various scenarios. For situations with limited computational resources or when the objective is to incorporate Medusa into an existing model without affecting its quality, we recommend Medusa-1. This method requires minimal memory and can be further optimized with quantization techniques akin to those in QLoRA [214], without compromising the generation quality due to the fixed backbone model. However, in Medusa-1, the full potential of the backbone model is not utilized. We can further

fine-tune it to enhance the prediction accuracy of Medusa heads, which can directly lead to a greater speedup. Therefore, we introduce Medusa-2, which is suitable for scenarios with ample computational resources or for direct Supervised Fine-Tuning (SFT) from a base model. The key to Medusa-2 is a training protocol that enables joint training of the Medusa heads and the backbone model without compromising the model's next-token prediction capability and output quality. We propose different strategies for obtaining the training datasets depending on the model's training recipe and dataset availability. When the model is fine-tuned on a public dataset, it can be directly used for Medusa. If the dataset is unavailable or the model underwent a Reinforcement Learning with Human Feedback (RLHF) [215] process, we suggest a self-distillation approach to generate a training dataset for the Medusa heads.

Our experiments primarily focus on scenarios with a batch size of one, which is representative of the use case where LLMs are locally hosted for personal use. We test Medusa on models of varying sizes and training settings, including Vicuna-7B, 13B (trained with a public dataset), Vicuna-33B [216] (trained with a private dataset), and Zephyr-7B (trained with both supervised fine-tuning and alignment). Medusa can achieve a speedup of 2.3 to 2.8 times across different prompt types without compromising on the quality of generation.

Moreover, we explored the hardware constraints, specifically memory-bandwidth bound, and their impact on Medusa. By profiling the performance of FLOP/s vs. Operational Intensity across various GPUs, we examined the changes when using Medusa for different operators. We propose a straightforward analytical model that allows us to predict the acceleration rates of Medusa, providing insights into the effects under different model sizes, sequence lengths, and batch sizes on target devices.

6.2 Related Work

6.2.1 LLM Inference Acceleration

The inefficiency of Large Language Model (LLM) inference is primarily attributed to the memory-bandwidth-bound nature of the auto-regressive de-

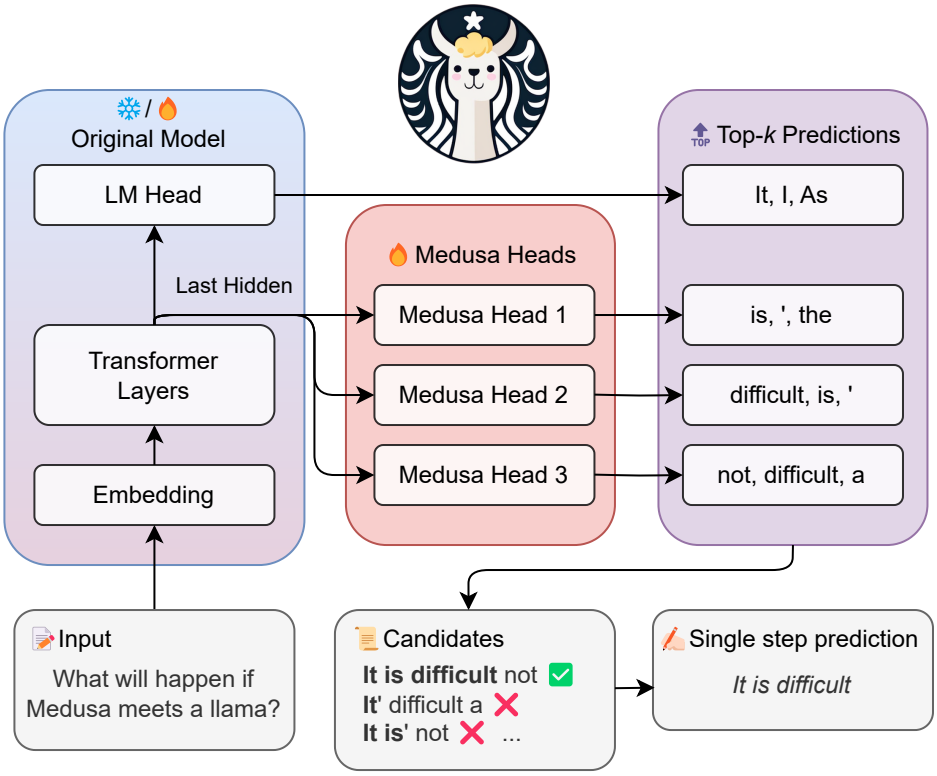


Figure 6.1: Medusa introduces *multiple heads* on top of the last hidden states of the LLM, enabling the prediction of several subsequent tokens in parallel. During inference, each head generates multiple top predictions for its designated position. These predictions are assembled into candidates, which are processed in parallel using a *tree-based attention* mechanism. The final step is to verify the candidates and accept a continuation. Besides the standard rejection sampling scheme, a *typical acceptance* scheme can also be used here to select reasonable continuations, and the *longest accepted candidate prefix* will be used for the next decoding phase.

coding process. Several methods have been proposed to alleviate this issue, improving inference latency and throughput. Traditionally, batch inference has been employed as a straightforward method to enhance arithmetic intensity and escape memory-bandwidth-bound limitations. However, with LLMs, both model parameters and the Key-Value (KV) cache consume substantial accelerator memory, hindering the utilization of large batch sizes. Existing methods to tackle this problem can be conceptually divided into two main categories: (1) Reducing memory consumption, thereby minimizing memory transfer overhead and enabling larger batch sizes, and (2) Minimizing the number of decoding steps to decrease latency directly.

Reducing KV Cache. Methods such as Multi-query attention [207] and Grouped-query attention [217] adopt a direct approach to diminish the KV cache. By utilizing fewer key and value heads in the attention modules relative to query heads, these strategies substantially cut the KV's memory consumption, thereby facilitating larger batch sizes and enhanced accelerator utilization [218]. Additionally, [219] proposes to selectively retain the most critical KV tokens, further reducing the KV cache. From a system perspective, [220] introduces a paged memory management scheme for reducing fragmentation of the KV cache.

Quantization. Quantization techniques are extensively used to shrink LLMs' memory consumption. Xiao et al. [221] apply rescaling between activations and parameters to eliminate outliers and simplify the quantization process. Dettmers et al. [222] breaks down matrix multiplications into predominantly 8-bit and a minority of 16-bit operations. Frantar et al. [223] iteratively round weight columns into 3/4 bits, while Lin et al. [224] present an activation-aware quantization scheme to protect salient weights and compress LLMs to 3/4 bits. Kim et al. [208] introduce a sparse plus low-precision pattern to handle a minor portion of vital weights, among other techniques.

Speculative Decoding. As an approach orthogonal to the aforementioned methods, speculative decoding [209, 210] aims to execute several decoding steps in parallel, thus reducing the total number of steps required. This parallelization is realized by employing a smaller draft model to conjecture several subsequent words, which the LLMs then collectively evaluate and accept as appropriate. While resonating with non-autoregressive generation literature [225], this method is specifically tailored for LLMs to address the aforementioned inefficiency. Unlike previous works, we propose leveraging the original model to make predictions rather than introducing an additional draft model. This approach is more straightforward and seamlessly integrates into existing systems without the complexities of managing two models. Independently, works[212, 226] propose the use of tree-structured attention to generate multiple candidates in parallel, where Miao el al.[212] suggest employing an ensemble of models to propose candidates, and Spector et al.[226] advocate adding another hierarchy for the draft model. However, draft models require specialized pretraining and alignment with the target models.

While employing multiple draft models can be cumbersome and involves the complexity of managing parallelism, our approach, which relies solely on decoding heads, offers a simpler alternative. Miao et al. [212] employ multiple draft models to generate tokens and merge them using tree attention, while Spector et al. [226] utilize a small draft model to process each level of the tree in batches. In contrast, our method directly uses the top predicted tokens from each of Medusa heads to create a static sparse tree without autoregression or adjusting the tree structure. This approach simplifies the process and improves efficiency. Additionally, we demonstrate through a detailed ablation study how the nodes of the tree can affect decoding speed.

6.2.2 Sampling Scheme

The manner in which text is sampled from Large Language Models (LLMs) can significantly influence the quality of the generated output. Recent studies have revealed that direct sampling from a language model may lead to incoherent or nonsensical results [227, 228]. In response to this challenge, *truncation sampling* schemes have been introduced [229, 230, 231, 232, 233]. These approaches aim to produce high-quality and diverse samples by performing sampling on a truncated distribution over a specific *allowed set* at each decoding step.

Different strategies define this allowed set in various ways. For example, top-k sampling [229] retains the k most likely words, whereas top-p sampling [228] incorporates the minimal set of words that account for p percent of the probability. Another method, known as typical decoding [233], employs the entropy of the predicted distribution to establish the threshold for inclusion. Hewitt et al. [232] offers a unified framework to understand truncation sampling techniques comprehensively.

Drawing inspiration from these methods, our typical acceptance scheme aligns with the concept of defining an allowed set to exclude improbable candidates from the sampling process. However, we diverge because we do not insist on an exact correspondence between the output and language model distribution. This deviation allows us to facilitate more diverse yet high-quality outputs, achieving greater efficiency without compromising the integrity of the generated text.

6.3 Methodology

Medusa follows the same framework as speculative decoding, where each decoding step primarily consists of three substeps: (1) generating candidates, (2) processing candidates, and (3) accepting candidates. For Medusa, (1) is achieved by Medusa heads, (2) is realized by tree attention, and since Medusa heads are on top of the original model, the logits calculated in (2) can be used for substep (1) for the next decoding step. The final step (3) can be realized by either rejection sampling [209, 210] or typical acceptance. The overall pipeline is illustrated in Figure 6.1.

In this section, we first introduce the key components of Medusa, including Medusa heads, and tree attention. Then, we present two levels of fine-tuning procedures for Medusa to meet the needs of different use cases. Finally, we propose two extensions to Medusa, including an automated self-distillation and typical acceptance, to handle situations where no training data is available for Medusa and to improve the efficiency of the decoding process, respectively.

6.3.1 Key Components

6.3.1.1 Medusa Heads

In speculative decoding, subsequent tokens are predicted by an auxiliary draft model. This draft model must be small yet effective enough to generate continuations that the original model will accept. Fulfilling these requirements is a challenging task, and existing approaches [226, 212] often resort to separately *pre-training* a smaller model. This pre-training process demands substantial additional computational resources. For example, in the work [212], a reported 275 NVIDIA A100 GPU hours were used. Additionally, separate pre-training can potentially create a distribution shift between the draft model and the original model, leading to continuations that the original model may not favor. Chen et al. [210] have also highlighted the complexities of serving multiple models in a distributed environment.

To streamline and democratize the acceleration of LLM inference, we take inspiration from Stern et al. [213], which utilizes parallel decoding for tasks such as machine translation and image super-resolution. Medusa heads are

additional decoding heads appended to the last hidden states of the original model. Specifically, given the original model's last hidden states h_t at position t, we add K decoding heads to h_t. The k-th head is used to predict the token in the $(t+k+1)$-th position of the next tokens (the original language model head is used to predict the $(t+1)$-th position). The prediction of the k-th head is denoted as $p_t^{(k)}$, representing a distribution over the vocabulary, while the prediction of the original model is denoted as $p_t^{(0)}$. Following the approach of [213], we utilize a single layer of feed-forward network with a residual connection for each head. We find that this simple design is sufficient to achieve satisfactory performance. The definition of the k-th head is outlined as:

$$p_t^{(k)} = \text{softmax}\left(W_2^{(k)} \cdot \left(\text{SiLU}(W_1^{(k)} \cdot h_t) + h_t\right)\right), \\ \text{where } W_2^{(k)} \in \mathbb{R}^{d \times V}, W_1^{(k)} \in \mathbb{R}^{d \times d}.$$

d is the output dimension of the LLM's last hidden layer and V is the vocabulary size. We initialize $W_2^{(k)}$ identically to the original language model head, and $W_1^{(k)}$ to zero. This aligns the initial prediction of the Medusa heads with that of the original model. The SiLU activation function [234] is employed following the Llama models [20].

Unlike a draft model, Medusa heads are trained in conjunction with the original backbone model, which can remain *frozen* during training (Medusa-1) or be trained together (Medusa-2). This method allows for fine-tuning large models even on a single GPU, taking advantage of the powerful base model's learned representations. Furthermore, it ensures that the distribution of the Medusa heads aligns with that of the original model, thereby mitigating the distribution shift problem. Additionally, since the new heads consist of just a single layer akin to the original language model head, Medusa does not add complexity to the serving system design and is friendly to distributed settings. We will discuss the training recipe for Medusa heads in Section 6.3.2.

6.3.1.2 Tree Attention

Through Medusa heads, we obtain probability predictions for the subsequent $K+1$ tokens. These predictions enable us to create length-$K+1$ continuations as candidates. While the speculative decoding studies [209, 210] suggest sampling a single continuation as the candidate, leveraging multiple candidates during decoding can enhance the expected acceptance length within a decoding step. Nevertheless, more candidates can also raise computational demands. To strike a balance, we employ a tree-structured attention mechanism to process multiple candidates concurrently. This attention mechanism

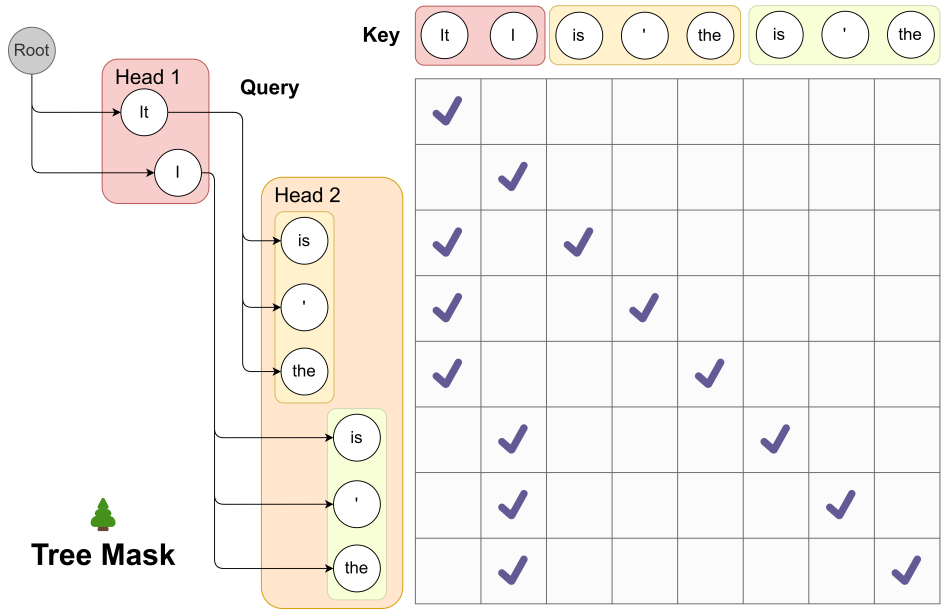


Figure 6.2: We demonstrates the use of tree attention to process multiple candidates concurrently. As exemplified, the top-2 predictions from the first Medusa head and the top-3 from the second result in a total of $2 \times 3 = 6$ candidates. Each of these candidates corresponds to a distinct branch within the tree structure. To guarantee that each token only accesses its predecessors, we devise an attention mask that exclusively permits attention flow from the current token back to its antecedent tokens. The positional indices for positional encoding are adjusted in line with this structure.

diverges from the traditional causal attention paradigm. Within this framework, only tokens from the same continuation are regarded as historical data. Drawing inspiration from the concept of embedding graph structures into attention as proposed in the graph neural network domain [235], we incorporate the tree structure into our attention mask, visualized in Figure 6.2. Remark-

ably, similar ideas have also been explored in independent works [212, 226], where they follow a bottom-up approach and construct the tree by merging multiple candidates generated by a draft model. In our method, we instead take a top-down approach to build the tree thanks to the structure of candidates generated by Medusa heads. For a given k-th head, its top-s_k predictions serve as the basis for candidate formation, where s_k is a designated hyperparameter. These candidates are established by determining the Cartesian product of the top-s_k predictions from each head. For instance, in Figure 6.2, with $s_1 = 2$ and $s_2 = 3$, each first head prediction can be succeeded by any prediction from the second head. This leads to a tree structure where s_k branches exist at the k-th level (considering a virtual root as the 0-level, in practice, this 0-level is for the prediction of the language model head of the original model, which can be sampled independently). Within this tree, only a token's predecessors are seen as historical context, and our attention mask ensures that the attention is only applied on a token's predecessors. By employing this mask and properly setting the positional indices for positional encoding, we can process numerous candidates simultaneously without the need to expand the batch size. The cumulative number of new tokens is calculated as $\sum_{k=1}^{K} \prod_{i=1}^{k} s_i$.

In this section, we demonstrate the most simple and regular way to construct the tree structure by taking the Cartesian product. However, it is possible to construct the tree structure in a more sophisticated way and exploit the unbalanced accuracy of different top predictions of different heads. We will discuss this in Section 6.3.3.

6.3.2 Training Strategies

At the most basic level, we can train Medusa heads by freezing the backbone model and fine-tune Medusa heads. However, training the backbone in conjunction with the Medusa heads can significantly enhance the accuracy of the Medusa heads. Depending on the computational resources and the specific requirements of the use case, we propose two levels of training strategies for Medusa heads.

In this section, we assume the availability of a training dataset that aligns with the target model's output distribution. This could be the dataset used

for Supervised Fine-Tuning (SFT) of the target model. We will discuss how to eliminate the need for such a dataset using a self-distillation approach in Section 6.3.3.

6.3.2.1 Medusa-1: Frozen Backbone

To train Medusa heads with a frozen backbone model, we can use the cross-entropy loss between the prediction of Medusa heads and the ground truth. Specifically, given the ground truth token y_{t+k+1} at position $t+k+1$, the loss for the k-th head is $\mathcal{L}_k = -\log p_t^{(k)}(y_{t+k+1})$ where $p_t^{(k)}(y)$ denotes the probability of token y predicted by the k-th head. We also observe that $\mathcal{L}_k$ is larger when k is larger, which is reasonable since the prediction of the k-th head is more uncertain when k is larger. Therefore, we can add a weight λ_k to $\mathcal{L}_k$ to balance the loss of different heads. And the total Medusa loss is:

$$\mathcal{L}_{\text{Medusa-1}} = \sum_{k=1}^{K} -\lambda_k \log p_t^{(k)}(y_{t+k+1}). \tag{6.1}$$

In practice, we set λ_k as the k-th power of a constant like 0.8. Since we only use the backbone model for providing the hidden states, we can use a quantized version of the backbone model to reduce the memory consumption. This introduces a more democratized way to accelerate LLM inference, as with the quantization, Medusa can be trained for a large model on a single consumer GPU similar to QLoRA [214]. The training only takes a few hours (e.g., 5 hours for Medusa-1 on Vicuna 7B model with a single NVIDIA A100 PCIE GPU to train on 60k ShareGPT samples).

6.3.2.2 Medusa-2: Joint Training

To further improve the accuracy of Medusa heads, we can train Medusa heads together with the backbone model. However, this requires a special training recipe to preserve the backbone model's next-token prediction capability and output quality. To achieve this, we propose three strategies:

- **Combined loss**: To keep the backbone model's next-token prediction capability, we need to add the cross-entropy loss of the backbone model $\mathcal{L}_{\text{LM}} = -\log p_t^{(0)}(y_{t+1})$ to the Medusa loss. We also add a weight

λ_0 to balance the loss of the backbone model and the Medusa heads. Therefore, the total loss is:

$$\mathcal{L}_{\text{Medusa-2}} = \mathcal{L}_{\text{LM}} + \lambda_0 \mathcal{L}_{\text{Medusa-1}}. \tag{6.2}$$

- **Differential learning rates**: Since the backbone model is already well-trained and the Medusa heads need more training, we can use separate learning rates for them to enable faster convergence of Medusa heads while preserving the backbone model's capability.
- **Heads warmup**: Noticing that at the beginning of training, the Medusa heads have a large loss, which leads to a large gradient and may distort the backbone model's parameters. Following the idea from [236], we can employ a two-stage training process. In the first stage, we only train the Medusa heads as Medusa-1. In the second stage, we train the backbone model and Medusa heads together with a warmup strategy. Specifically, we first train the backbone model for a few epochs, then train the Medusa heads together with the backbone model. Besides this simple strategy, we can also use a more sophisticated warmup strategy by gradually increasing the weight λ_0 of the backbone model's loss. We find both strategies work well in practice.

Putting these strategies together, we can train Medusa heads together with the backbone model without hurting the backbone model's capability. Moreover, this recipe can be applied together with Supervised Fine-Tuning (SFT), enabling us to get a model with native Medusa support.

6.3.3 Extensions

6.3.3.1 Typical Acceptance

In speculative decoding papers [209, 210], authors employ rejection sampling to yield diverse outputs that align with the distribution of the original model. However, subsequent implementations [237, 226] reveal that this sampling strategy results in diminished efficiency as the sampling temperature increases. Intuitively, this can be comprehended in the extreme instance where the draft model is the same as the original one. Here, when using

greedy decoding, all output of the draft model will be accepted, therefore maximizing the efficiency. Conversely, rejection sampling introduces extra overhead, as the draft model and the original model are sampled independently. Even if their distributions align perfectly, the output of the draft model may still be rejected.

However, in real-world scenarios, sampling from language models is often employed to generate diverse responses, and the temperature parameter is used merely to modulate the "creativity" of the response. Therefore, higher temperatures should result in more opportunities for the original model to accept the draft model's output. We ascertain that it is typically unnecessary to match the distribution of the original model. Thus, we propose employing a *typical acceptance* scheme to select plausible candidates rather than using rejection sampling. This approach draws inspiration from truncation sampling studies [232] (refer to Section 6.2.2 for an in-depth explanation). Our objective is to choose candidates that are *typical*, meaning they are not exceedingly improbable to be produced by the original model. We use the prediction probability from the *original model* as a natural gauge for this and establish a threshold based on the prediction distribution to determine acceptance. Specifically, given $x_1, x_2, \cdots, x_n$ as context, when evaluating the candidate sequence $(x_{n+1}, x_{n+2}, \cdots, x_{n+K+1})$ (composed by top predictions of the original language model head and Medusa heads), we consider the condition

$$p_{\text{original}}(x_{n+k}|x_1, x_2, \cdots, x_{n+k-1}) > \min\left(\epsilon, \delta \exp\left(-H(p_{\text{original}}(\cdot|x_1, x_2, \cdots, x_{n+k-1}))\right)\right),$$

where $H(\cdot)$ denotes the entropy function, and ϵ, δ are the hard threshold and the entropy-dependent threshold respectively. This criterion is adapted from [232] and rests on two observations: (1) tokens with relatively high probability are meaningful, and (2) when the distribution's entropy is high, various continuations may be deemed reasonable. During decoding, every candidate is evaluated using this criterion, and a *prefix* of the candidate is accepted if it satisfies the condition. To guarantee the generation of at least one token at each step, we apply *greedy decoding* for the first token and *unconditionally* accept it while employing typical acceptance for subsequent tokens. The final prediction for the current step is determined by the *longest*

accepted prefix among all candidates.

Examining this scheme leads to several insights. Firstly, when the temperature is set to 0, it reverts to greedy decoding, as only the most probable token possesses non-zero probability. As the temperature surpasses 0, the outcome of greedy decoding will consistently be accepted with appropriate ϵ, δ, since those tokens have the maximum probability, yielding maximal speedup. Likewise, in general scenarios, an increased temperature will correspondingly result in longer accepted sequences, as corroborated by our experimental findings.

Empirically, we verify that typical acceptance can achieve a better speedup while maintaining a similar generation quality as shown in Figure 6.7.

6.3.3.2 Automated Self-Distillation

In Section 6.3.2, we assume the existence of a training dataset that matches the target model's output distribution. However, this is not always the case. For example, the model owners may only release the model without the training data, or the model may have gone through a Reinforcement Learning with Human Feedback (RLHF) procedure, which makes the output distribution of the model different from the training dataset. To tackle this issue, we propose an automated self-distillation pipeline to use the model itself to generate the training dataset for Medusa heads, which matches the output distribution of the model.

The dataset generation process is straightforward. We first take a public seed dataset from a domain similar to the target model; for example, using the ShareGPT [238] dataset for chat models. Then, we simply take the prompts from the dataset and ask the model to reply to the prompts. In order to obtain multi-turn conversation samples, we can sequentially feed the prompts from the seed dataset to the model. Or, for models like Zephyr 7B [239], which are trained on both roles of the conversation, they have the ability to self-talk, and we can simply feed the first prompt and let the model generate multiple rounds of conversation.

For Medusa-1, this dataset is sufficient for training Medusa heads. However, for Medusa-2, we observe that solely using this dataset for training the backbone and Medusa heads usually leads to a lower generation quality. In fact, even without training Medusa heads, training the backbone model with

this dataset will lead to quality degradation. This suggests that we also need to use the original model's probability prediction instead of using the ground truth token as the label for the backbone model, similar to classic knowledge distillation works [240]. Concretely, the loss for the backbone model is:

$$\mathcal{L}_{\text{LM-distill}} = KL(p^{(0)}_{\text{original},t} || p^{(0)}_t),$$

where $p^{(0)}_{\text{original},t}$ denotes the probability distribution of the original model's prediction at position t.

However, naively, to obtain the original model's probability prediction, we need to maintain two models during training, increasing the memory requirements. To further alleviate this issue, we propose a simple yet effective way to exploit the self-distillation setup. We can use a parameter-efficient adapter like LoRA [241] for fine-tuning the backbone model. In this way, the original model is simply the model with the adapter turned off. Therefore, the distillation does not require additional memory consumption. Together, this self-distillation pipeline can be used to train Medusa-2 without hurting the backbone model's capability and introduce almost no additional memory consumption. Lastly, one tip about using self-distillation is that it is preferable to use LoRA without quantization in this case, otherwise, the teacher model will be the quantized model, which may lead to a lower generation quality.

6.3.3.3 Searching for the Optimized Tree Construction

In Section 6.3.1, we present the simplest way to construct the tree structure by taking the Cartesian product. However, with a fixed number of total nodes in the tree, a regular tree structure may not be the best choice. Intuitively, those candidates composed of the top predictions of different heads may have different accuracies. Therefore, we can leverage an estimation of the accuracy to construct the tree structure.

Specifically, we can use a calibration dataset and calculate the accuracies of the top predictions of different heads. Let $a^{(i)}_k$ denote the accuracy of the i-th top prediction of the k-th head. Assuming the accuracies are independent, we can estimate the accuracy of a candidate sequence composed by the top $[i_1, i_2, \cdots, i_k]$ predictions of different heads as $\prod_{j=1}^{k} a^{(i_j)}_j$. Let I denote the

set of all possible combinations of $[i_1, i_2, \cdots, i_k]$ and each element of I can be mapped to a node of the tree (not only leaf nodes but all nodes are included). Then, the expectation of the acceptance length of a candidate sequence is:

$$\sum_{[i_1,i_2,\cdots,i_k]\in I} \prod_{j=1}^{k} a_j^{(i_j)}.$$

Thinking about building a tree by adding nodes one by one, the contribution of a new node to the expectation is exactly the accuracy associated with the node. Therefore, we can greedily add nodes to the tree by choosing the node that is connected to the current tree and has the highest accuracy. This process can be repeated until the total number of nodes reaches the desired number. In this way, we can construct a tree that maximizes the expectation of the acceptance length.

Figure 6.3 illustrates the structure of a sparsely constructed tree for the Medusa-2 Vicuna-7B model. This tree structure extends four levels deep, indicating the engagement of four Medusa heads in the computation. The tree is initially formed through a Cartesian product approach and subsequently refined by pruning based on the statistical expectations of the top-k predictions from each Medusa head measured on the Alpaca-eval dataset [242]. The tree's lean towards the left visually represents the algorithm's preference for nodes with higher probabilities on each head.

6.4 Experiments

In this section, we present experiments to demonstrate the effectiveness of Medusa in different settings. First, we evaluate Medusa on the Vicuna-7B and 13B models [216] to show the performance of Medusa-1 and Medusa-2. Then, we assess our method using the Vicuna-33B and Zephyr-7B models to demonstrate self- distillation's viability in scenarios where direct access to the fine-tuning recipe is unavailable, as with Vicuna-33B, and in models like Zephyr-7B that employ Reinforcement Learning from Human Feedback (RLHF). The evaluation is conducted on MT-Bench [243], a multi-turn, conversational-format benchmark.

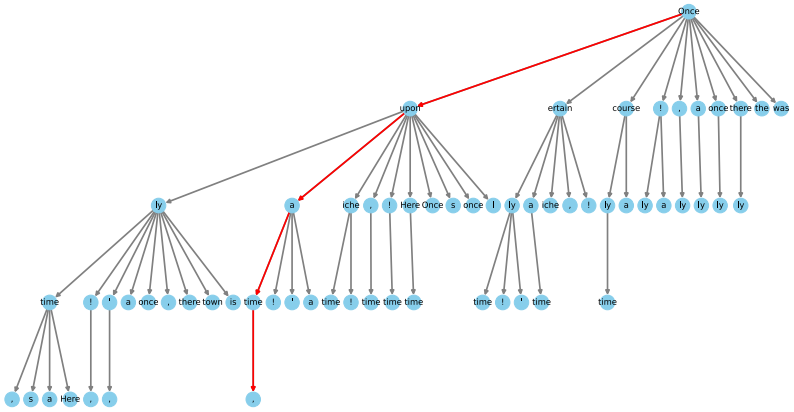


Figure 6.3: Visualization of a sparse tree setting for Medusa-2 Vicuna-7B. The tree has 64 nodes representing candidate tokens and a depth of 4 which indicates 4 Medusa heads involved in calculation. Each node indicates a token from a top-k prediction of a Medusa head, and the edges show the connections between them. The red lines highlight the path that correctly predicts the future tokens.

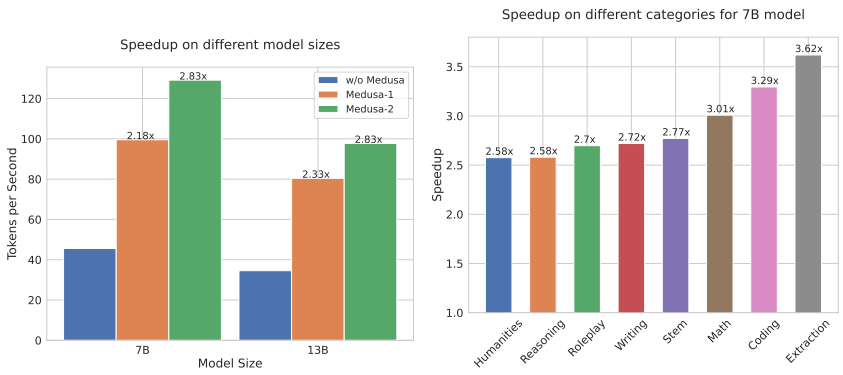
6.4.1 Experiment Settings

6.4.1.1 Common Terms

We clarify three commonly used terms: a) Acceleration rate: This refers to the average number of tokens decoded per decoding step. In a standard auto-regressive model, this rate is 1.0. b) Overhead: This is used to characterize the per decoding step overhead compared to classic decoding, and is calculated by dividing the average per step latency of the Medusa models by that of the vanilla model. c) Speedup: This refers to the wall-time acceleration rate. Following these definitions, we have the relation: Speedup = Acceleration rate / Overhead.

6.4.1.2 Shared Settings

For all the experiments, we use the Axolotl [244] framework for training. We use a cosine learning rate scheduler with warmup and use 8-bit AdamW [245] optimizer. We train five Medusa heads with one layer and set λ_k in Eq. (6.1) to be 0.8^k. For Medusa-2, we use either LoRA [241] or QLoRA [214] for fine-tuning and set the learning rate of Medusa heads to be 4 times larger than the backbone model. LoRA is applied to all the linear layers of the backbone



(a) (b)

Figure 6.4: Left: Speed comparison of baseline, Medusa-1 and Medusa-2 on Vicuna-7B/13B. Medusa-1 achieves more than 2× wall-time speedup compared to the baseline implementation while Medusa-2 further improves the speedup by a significant margin. Right: Detailed speedup performance of Vicuna-7B with Medusa-2 on 8 categories from MT-Bench.

model, including the language model head. The rank of LoRA adapter is set to 32, and α is set to 16. A dropout of 0.05 is added to the LoRA adapter.

6.4.1.3 Medusa-1 v.s. Medusa-2 on Vicuna 7B and 13B

We use a global batch size of 64 and a peak learning rate of $5e^{-4}$ for the backbone and $2e^{-3}$ for Medusa heads and warmup for 40 steps. We use 4-bit quantized backbone models for both models. We first train the models with Medusa-1 and use these trained models as initialization to train Medusa-2. We employ QLoRA for Medusa-2 and the λ_0 in Eq. (6.2) is set to be 0.2.

6.4.1.4 Training with Self-Distillation on Vicuna-33B and Zephyr-7B

We use Medusa-2 for both models and instead of using a two-stage training procedure, we use a sine schedule for the θ_0 to gradually increase the value to its peak at the end of the training, we find this approach is equally effective. We set the peak learning rate of the backbone LoRA adapter to be $1e^{-4}$ and the warmup steps to be 20. Since the self-distillation loss is relatively small, we set the λ_0 in Eq. (6.2) to be 0.01.

6.4.2 Performance Evaluation of Medusa-1 and Medusa-2

In this section, we present the experimental results comparing Medusa-1 and Medusa-2 configurations on models ranging from 7B to 33B. Our experiments, conducted with a single batch size, focus on evaluating speedup, quality scores, and effectiveness across different tasks and datasets. We begin by showcasing the results and comparing Medusa configurations to existing speculative decoding strategies. Additionally, we explore the configurations of tree attention, thresholds for typical acceptance, and self-distillation, providing insights into their impact on performance.

6.4.2.1 Case Study: Medusa-1 v.s. Medusa-2 on Vicuna 7B and 13B

Experimental Setup. We use the Vicuna model class [216], which encompasses chat models of varying sizes (7B, 13B, 33B) that are fine-tuned from the Llama model [20]. Among them, the 7B and 13B models are trained on the ShareGPT [238] dataset, while the 33B model is an experimental model and is trained on a private dataset. In this section, we use the ShareGPT dataset to train the Medusa heads on the 7B and 13B models for 2 epochs. We use the v1.5 version of Vicuna models, which are fine-tuned from Llama-2 models with sequence length 4096.

Results. We collect the results and show them in Figure 6.4. The baseline is the default Huggingface implementation. In Figure 6.4a, we can see that for the 7B models, Medusa-1 and Medusa-2 configurations lead to a significant increase in speed, measuring in tokens processed per second. Medusa-1 shows a 2.18× speedup, while Medusa-2 further improves this to a 2.83×. When applied to the larger 13B model, Medusa-1 results in a 2.33× speed increase, while Medusa-2 maintains a similar performance gain of 2.83× over the baseline. We also plot the speedup per category for the Medusa-2 Vicuna-7B model. We observe that the coding category benefits from a 3.29× speedup, suggesting that Medusa is particularly effective for tasks in this domain. This points to a significant potential for optimizing coding LLMs, widely used in software development and other programming-related tasks. The "Extraction" category shows the highest speedup at 3.62×, indicating that this task is highly optimized by the Medusa. Overall, the results suggest that the Medusa significantly enhances inference speed across different model sizes

and tasks.

6.4.2.2 Case Study: Training with Self-Distillation on Vicuna-33B and Zephyr-7B

Experimental Setup. In this case study, we focus on the cases where self-distillation is needed. We use the Vicuna-33B model [216] and the Zephyr-7B model [239] as examples. Following the procedure described in Section 6.3.3, we first generate the datasets with some seed prompts. We use ShareGPT [238] and UltraChat [246] as the seed datasets and collect a dataset at about $100k$ samples for both cases. Interestingly, we find that the Zephyr model can continue to generate multiple rounds of conversation with a single prompt, which makes it easy to collect a large dataset. For Vicuna-33B, we generate the multi-turn conversations by iteratively feeding the prompts from each multi-turn seed conversation. Both models are trained with sequence length 2048 and batch size 128.

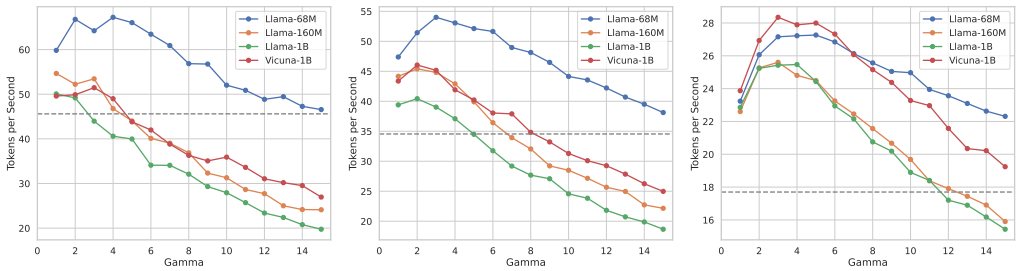
Results. Table 6.1 complements these findings by comparing various Medusa-2 models in terms of their acceleration rate, overhead, and quality on MT-Bench with GPT-4 acting as the evaluator to assign quality scores ranging from 0 to 10. We report the quality differences of Medusa compared to the original model. Notably, while the Medusa-2 Vicuna-33B model shows a lower acceleration rate, it maintains a comparable quality. We hypothesize that this is due to a mismatch between the hidden training dataset and the dataset we used for self-distillation.

6.4.2.3 Comparison with Existing Speculative Decoding Algorithms

In our study, we also applied speculative decoding [210, 209] to the Vicuna lineup using open-source draft models. The preliminary framework utilized open-source models such as Llama-68M and 160M [212], alongside Tiny-Llama [247] and Tiny-Vicuna [248], fine-tuned from Tiny-Llama with the Vicuna-style instructional tuning strategy. Due to the proprietary nature of speculative decoding methods [210, 209], open-source alternatives[1] were deployed for evaluation. Additionally, we utilize `torch.compile()` to accelerate the inference speed of draft models.

[1] https://github.com/feifeibear/LLMSpeculativeSampling

Our results shown in Figure 6.5, reveal that the optimal settings of the draft model vary with the Vicuna model sizes. Specifically, the Llama-68M, with a setting of $\gamma = 4$, yielded the best performance for Vicuna-7B, while the same draft model with $\gamma = 3$ was most effective for Vicuna-13B. For the larger Vicuna-33B, the Tiny-Vicuna, with $\gamma = 3$, provided the greatest acceleration. These results suggest that the choice and setting of the drafting model should be tailored to the size of the LLMs, presenting an area for further exploration in the field.



(a) Vicuna-7B (b) Vicuna-13B (c) Vicuna-33B

Figure 6.5: Inference speed of various models using speculative decoding on MT-Bench. Baseline model speeds are presented by grey dotted lines for comparison. γ denotes the draft token number.

These results underscore the complex interplay between speed and quality when scaling up model sizes and applying self-distillation techniques. The findings also highlight the potential of the Medusa-2 configuration to boost efficiency in processing while carefully preserving the quality of the model's outputs, suggesting a promising direction for co-optimizing LLMs with Medusa heads.

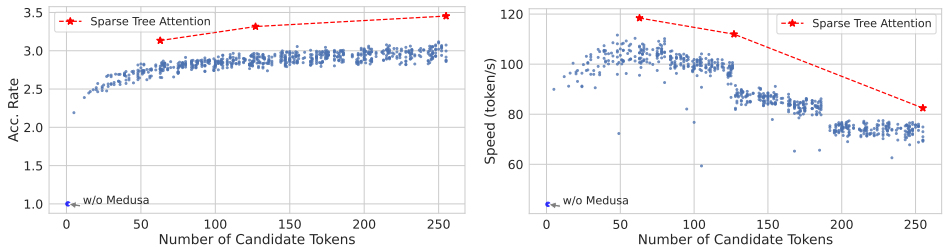
6.4.2.4 Ablation Studies on Extensions

Configuration of Tree Attention The study of tree attention is conducted on the writing and roleplay categories from the MT-Bench dataset using Medusa-2 Vicuna-7B. We target to depict tree attention's motivation and its performance.

Figure 6.6a compares the acceleration rate of randomly sampled dense tree configurations (depicted by blue dots) against optimized sparse tree settings (shown with red stars). The sparse tree configuration with 64 nodes shows a

Table 6.1: Comparison of various Medusa-2 models. The first section reports the details of Medusa-2, including accelerate rate, overhead, and quality that denoted the average scores on the MT-Bench compared to the original models. The second section lists the speedup (S) of SpecDecoding and Medusa, respectively.

Model Name	Vicuna-7B	Zephyr-7B	Vicuna-13B	Vicuna-33B
Acc. rate	3.47	3.14	3.51	3.01
Overhead	1.22	1.18	1.23	1.27
Quality	6.18 (+0.01)	7.25 (-0.07)	6.43 (-0.14)	7.18 (+0.05)
$S_{\text{SpecDecoding}}$	1.47	-	1.56	1.60
S_{Medusa}	2.83	2.66	2.83	2.35



(a) (b)

Figure 6.6: Effectiveness of numbers of candidate tokens for decoding introduced by trees (default number of candidate token for decoding is 1 when using KV cache). Left: The acceleration rate for randomly sampled dense tree settings (blue dots) and optimized sparse tree settings (red stars). Right: The speed (tokens/s) for both settings. The trend lines indicate that while the acceleration rate remains relatively stable for sparse trees, there is a notable decrease in speed as the candidate tokens increases.

better acceleration rate than the dense tree settings with 256 nodes. The decline in speed in Fig. 6.6b is attributed to the increased overhead introduced by the compute-bound. While a more complex tree can improve acceleration, it does so at the cost of speed due to intensive matrix multiplications for linear layers and self-attention. The acceleration rate increase follows a logarithmic trend and slows down when the tree size grows as shown in Fig. 6.6a. However, the initial gains are substantial, allowing Medusa to achieve significant speedups. If the acceleration increase is less than the overhead, it will slow down overall performance. For detailed study, please refer to Section 6.4.3.

Thresholds of Typical Acceptance The thresholds of typical acceptance are studied on the writing and roleplay categories from the MT-Bench

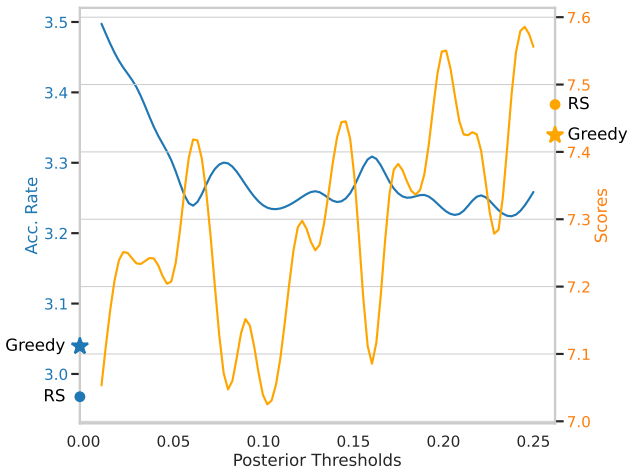


Figure 6.7: Performance comparison of Medusa using proposed typical sampling. The model is fully fine-tuned from Vicuna-7B. The plot illustrates the acceleration rate and average scores on the writing and roleplay (MT-Bench) with a fixed temperature of 0.7 for three different settings: greedy sampling and random sampling (RS) plotted as the star and the dot, and typical sampling curves under different thresholds.

dataset [243] using Medusa-2 Vicuna 7B. Utilizing the Vicuna 7B model, we aligned our methodology with the approach delineated by [232] setting the $\alpha = \sqrt{\epsilon}$. Figure 6.7 presents a comparative analysis of our model's performance across various sampling settings. These settings range from a threshold ϵ starting at 0.01 and incrementally increasing to 0.25 in steps of 0.01. Our observations indicate a discernible trade-off: as ϵ increases, there is an elevation in quality at the expense of a reduced acceleration rate. Furthermore, for tasks demanding creativity, it is noted that the default random sampling surpasses greedy sampling in performance, and the proposed typical sampling is comparable with random sampling when ϵ increases.

Table 6.2: Comparison of Different Settings of Vicuna-7B. Quality is obtained by evaluating models on MT-Bench using GPT-4 as the judge (higher the better).

	Baseline	Direct Fine-tuning	Medusa-1	Medusa-2
Quality	6.17	5.925	6.23	6.18
Speedup	N/A	N/A	2.18	2.83

Effectiveness of Two-stage Fine-tuning We examine the performance differences between two fine-tuning strategies for the Vicuna-7B model in

Table 6.2. We provided the comparison of directly fine-tuning the model with the Medusa heads vs. Medusa-2 that involves two-stage fine-tuning described in Section 6.3.2. The findings indicate that implementing our Medusa-2 for fine-tuning maintains the model's quality and concurrently improves the speedup vs. Medusa-1.

6.4.3 Exploration and Modeling of Hardware Constraints and Medusa

In the previous section, we evaluated the performance improvements brought by Medusa. However, hardware constraints present significant challenges, especially when dealing with large batch sizes that may lead to out-of-memory (OOM) errors if the entire model is loaded onto a GPU. Implementing efficient distributed GPU setups can also be complex and requires careful consideration.

We explore the hardware constraints, specifically memory-bandwidth bound, and their impact on Medusa-style parallel decoding by incorporating a simplified Llama-series model. First, we identify that the operators involving matrix multiplications, such as linear layers and attention matrix multiplications, are the primary sources of overhead. We profile the performance of FLOP/s vs. Operational Intensity which is the ratio of FLOP/s to bandwidth (bytes/s), across various GPUs, including the A100-80GB-PCIe, A40, and A6000. Next, we examine the changes in FLOP/s vs. Operational Intensity when using Medusa for different operators. Finally, we apply a straightforward analytical model to calculate acceleration rates and combine it with hardware benchmarks. This provides insights into the effects under different model sizes, sequence lengths, and batch sizes.

6.4.3.1 Roofline Model of Operators

We present an analysis of the roofline model for various operators in large language models (LLMs), specifically focusing on Llama-7B, Llama-13B, and Llama-33B [20]. These models were benchmarked on different GPUs, including the A100-80GB-PCIe, A40, and A6000. We looked into the three categories of matrix multiplication operators since they represent the primary

sources of computational overhead in these models. Our study follows the report [249] which investigates the effectiveness of batch size but ours focuses more on decoding and parallel decoding.

Table 6.3 details the computation and space complexity for each operator during the prefill, decoding, and Medusa decoding phases. The operators include the linear layers for query, key, and value matrices (XW_Q, XW_K, XW_V), the attention matrix multiplications (QK^T, PV), and the up/gate/-down linear layers (XW_u, XW_g, XW_d). b stands for the batch size, s stands for the sequence length, h stands for the hidden dimension, i stands for the intermediate dimension, n stands for the number of attention heads, d stands for the head dimension and q stands for the candidate length for Medusa. For more details of these operators please refer to the articles [20, 249].

Table 6.3: Computational and space complexity of the main operators in different phases. The table is based on Table 2 in the work [249].

Operator	Input Shape	Output Shape	Comp. Complexity	Space Complexity
Prefill				
XW_Q, XW_K, XW_V	(b, s, h)	(b, s, h)	$O(bsh^2)$	$O(2bsh + h^2)$
QK^T PV	$(b, n, s, d), (b, n, s, d)$ $(b, n, s, s), (b, n, s, d)$	(b, n, s, s) (b, n, s, d)	$O(bs^2nd)$	$O(2bsnd + bs^2n)$
XW_u, XW_g XW_d	(b, s, h) (b, s, i)	(b, s, i) (b, s, h)	$O(bshi)$	$O(bs(h + i) + hi)$
Decoding				
XW_Q, XW_K, XW_V	$(b, 1, h)$	$(b, 1, h)$	$O(bh^2)$	$O(2bh + h^2)$
QK^T PV	$(b, n, 1, d), (b, n, s, d)$ $(b, n, s, 1), (b, n, 1, d)$	$(b, n, s, 1)$ $(b, n, 1, d)$	$O(bsnd)$	$O(bsn + bsnd + bnd)$
XW_u, XW_g XW_d	$(b, 1, h)$ $(b, 1, i)$	$(b, 1, i)$ $(b, 1, h)$	$O(bhi)$	$O(b(h + i) + hi)$
Parallel decoding				
XW_Q, XW_K, XW_V	(b, q, h)	(b, q, h)	$O(bqh^2)$	$O(2bqh + h^2)$
QK^T PV	$(b, n, q, d), (b, n, s, d)$ $(b, n, s, q), (b, n, q, d)$	(b, n, s, q) (b, n, q, d)	$O(bsqnd)$	$O(bsqn + b(s + q)nd)$
XW_u, XW_g XW_d	(b, q, h) (b, q, i)	(b, q, i) (b, q, h)	$O(bqhi)$	$O(bq(h + i) + hi)$

Figures 6.8-6.16 show the benchmark of three categories of operators on different models (7/13/33B) under various settings. To evaluate each operator's performance and throughput, we chose the combination of settings including batch sizes from 1 to 64 in powers of 2 and sequence lengths from 128 to 8192 in powers of 2 (49 settings for each operator). From all the figures, we observe that the datapoints of each operator in the prefill and decoding stages cluster at very similar positions across all GPUs and for various model sizes.

During the prefill phase, increasing the batch size changes the FLOP/s of the attention matrix multiplications (see `'qk/pv init'`) but does not affect the Operational Intensity (refer to the vertical dashed arrow in Figure 6.8). In contrast, increasing the sequence length impacts both FLOP/s and Operational Intensity in the prefill phase (refer to the diagonal dashed arrow in Figure 6.8). During the decoding phase, the attention matrix multiplications are significantly limited by memory bandwidth. Despite an increase in FLOP/s with changes in batch size and sequence length, the Operational Intensity remains nearly unchanged (see `'qk/pv ar'`). This indicates suboptimal resource utilization in the self-attention mechanism.

The linear layers in the prefill phase are mostly compute-bound (see `'qkv mlp init'` and `'up/gate/down init'`). During the decoding phase, the datapoints of the linear layer form a line with the same slope as the GPU's memory bandwidth (see `'qkv mlp ar'` and `'up/gate/down ar'`). This indicates the linear layers in the decoding stage are also bounded by memory bandwidth. Increasing the batch size improves the achieved FLOP/s and Operational Intensity under memory bandwidth constraints through better parallelism. Note that linear layers only process the new token and are independent of sequence length (See 'Decoding' section in Table 6.3).

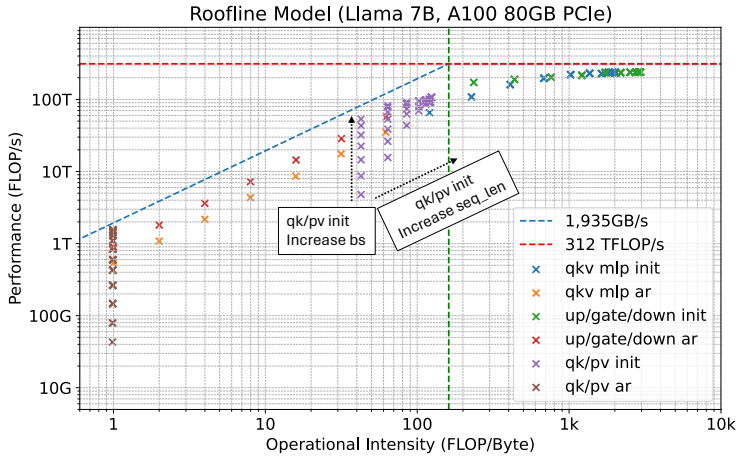


Figure 6.8: The figure shows the relationship between FLOP/s and Operational Intensity for all benchmarked datapoints of Llama-7B operators on A100-80GB-PCIe. The dashed lines represent the HBM bandwidth limit (1,935GB/s) and the peak performance limit (312 TFLOP/s) [250]. '`qkv mlp`' stands for the linear layers projecting hidden features to query/key/value features. '`up/gate/down`' stands for the linear layers following the attention block. '`qk/pv`' stands for the two steps of attention matrix multiplications. '`ar`' stands for the decoding (autoregressive) and '`init`' stands for the prefill phase.

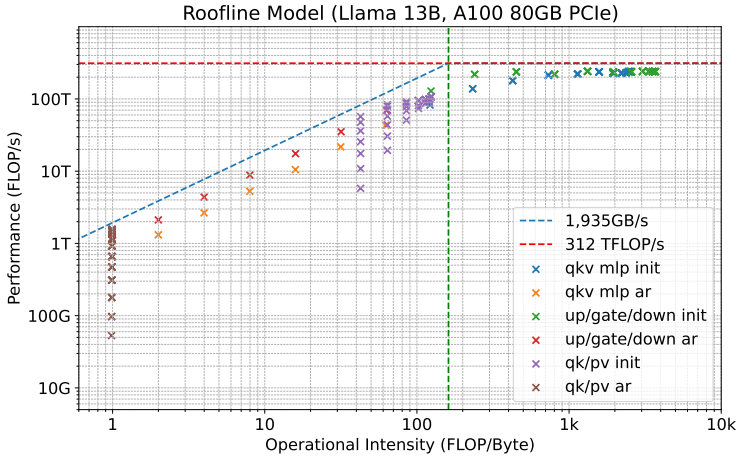


Figure 6.9: Llama-13B operators on A100-80GB-PCIe.

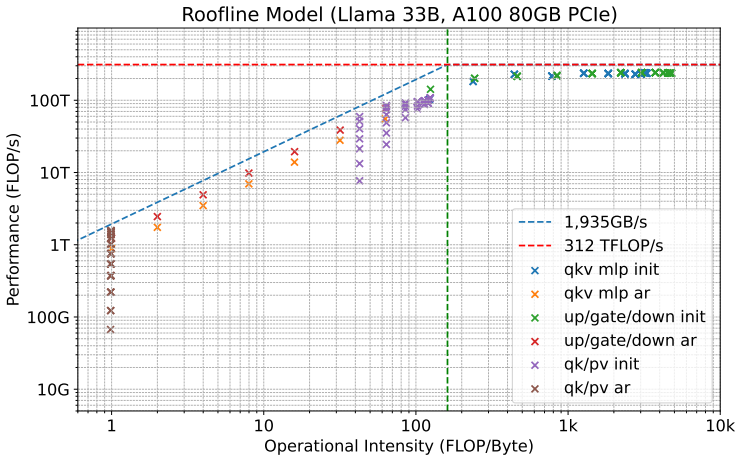


Figure 6.10: Llama-33B operators on A100-80GB-PCIe.

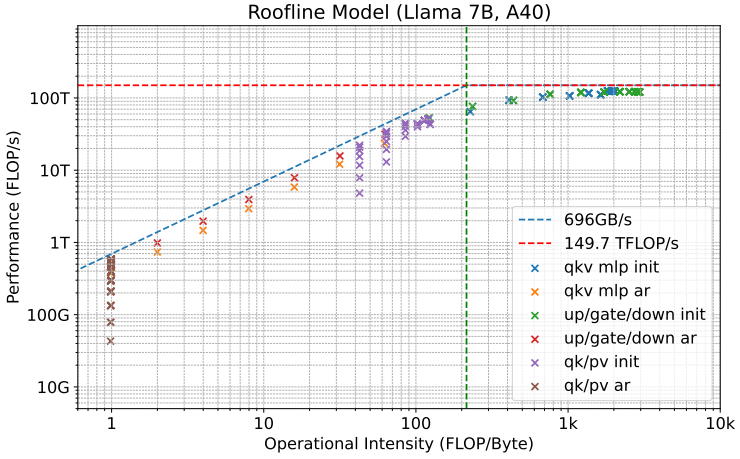


Figure 6.11: Llama-7B operators on A40.

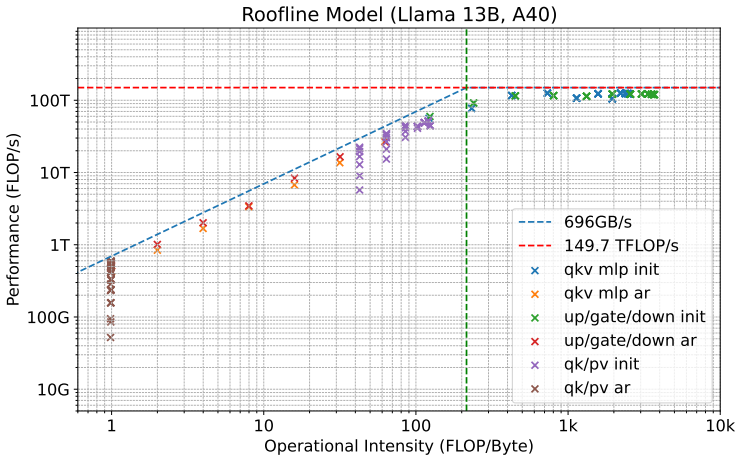


Figure 6.12: Llama-13B operators on A40.

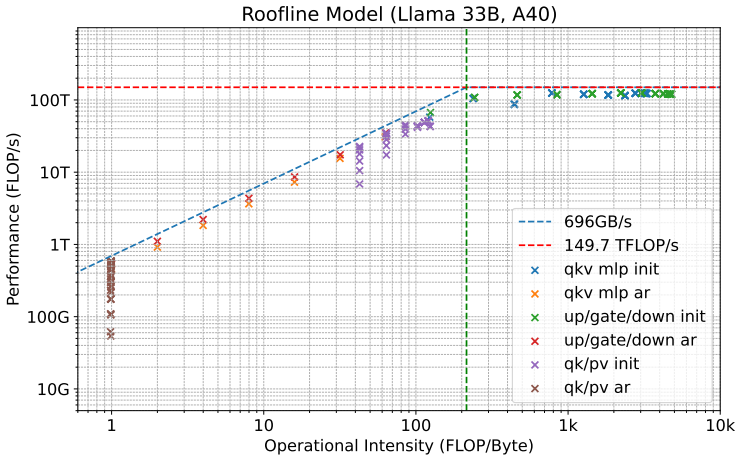


Figure 6.13: Llama-33B operators on A40.

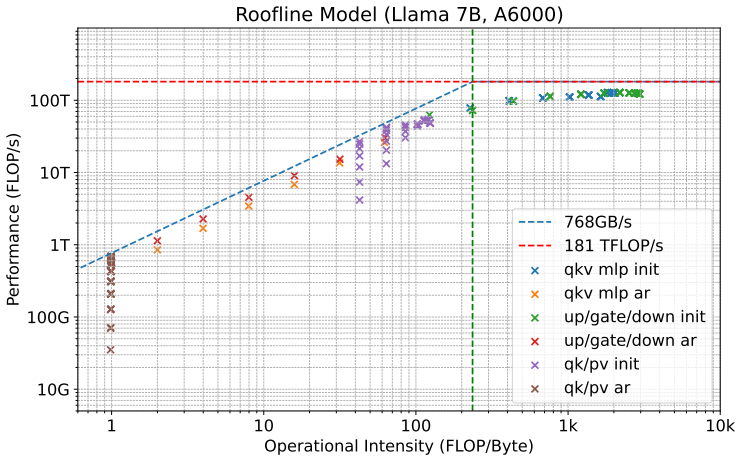


Figure 6.14: Llama-7B operators on A6000.

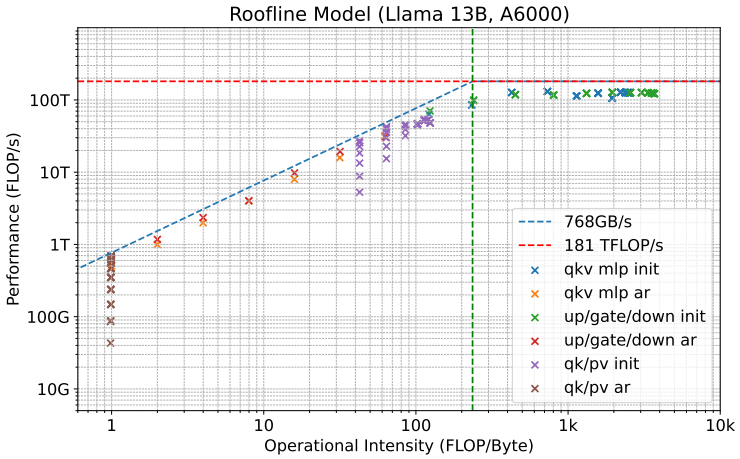


Figure 6.15: Llama-13B operators on A6000.

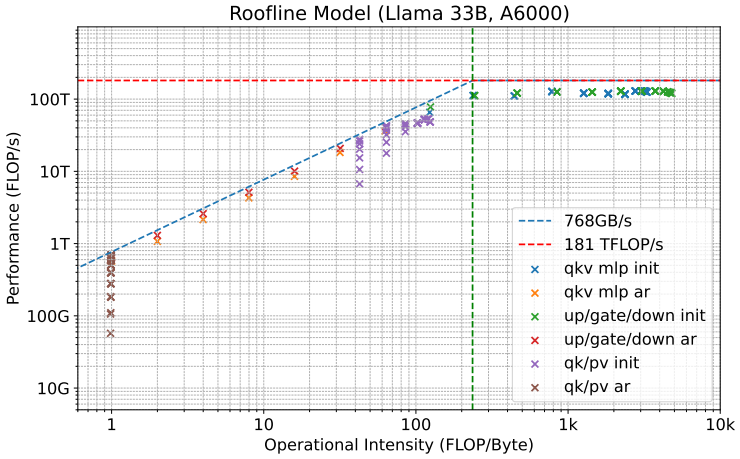


Figure 6.16: Llama-33B operators on A6000.

6.4.3.2 FLOP/s vs. Operational Intensity Variations in Medusa

We investigate how Medusa can change Operational Intensity and elevate the FLOP/s. We choose Llama 33B on A100-80GB-PCIe as the setting.

First, we examine the attention matrix multiplication. Figure 6.17 and Table 6.4 illustrate the effects of Medusa while keeping the batch size fixed at 16. We observe increased FLOP/s and Operational Intensity as more candidate tokens are added (original decoding results are plotted as grey dots). This indicates that Medusa can leverage additional candidate tokens to improve computational throughput. Compared to regular decoding, Medusa achieves 44× FLOP/s and 41× Operational Intensity under the setting of batch size 16 and sequence length 1024 with 64 candidate tokens. Figure 6.18 and Table 6.5 illustrate the effects of Medusa decoding while keeping the sequence length fixed at 1024. Increasing the batch size does not improve Operational Intensity in this scenario.

Next, we examine the linear layer, focusing on the up/gate/down linear layers. The results are shown in Figure 6.19 and Table 6.6. Since the linear layers in the decoding phase only process the future tokens while the past tokens are cached, they are independent of the sequence length. We vary the batch size to observe the effects. As Medusa increases the number of candidate tokens with the increasing batch size, we observe a shift from a memory-bandwidth-bound region to a computation-bound region. This shift demonstrates how Medusa can transition the performance characteristics of the linear layers from being limited by memory bandwidth to being limited by computational capacity.

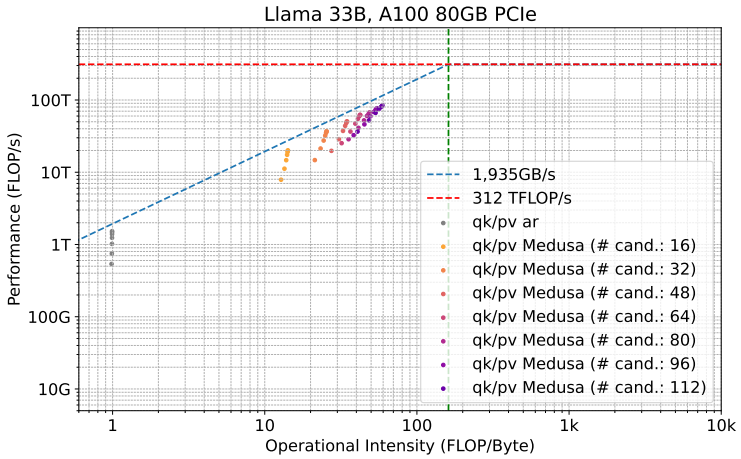


Figure 6.17: FLOP/s vs. operational intensity of attention matrix multiplication with batch size 16.

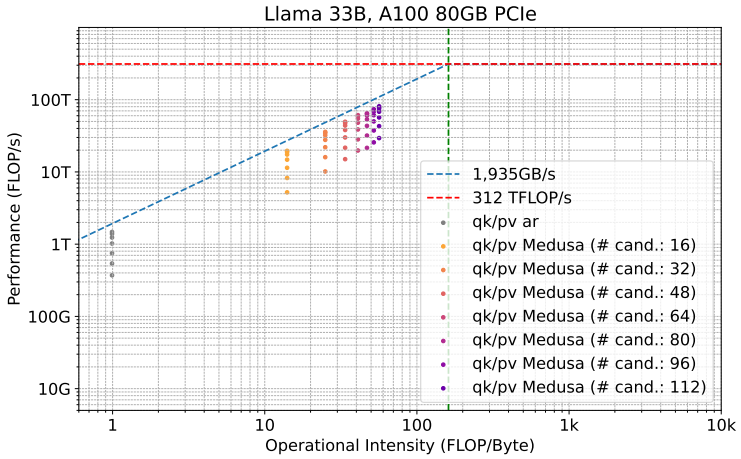


Figure 6.18: FLOP/s vs. operational intensity of attention matrix multiplication with sequence length 1024.

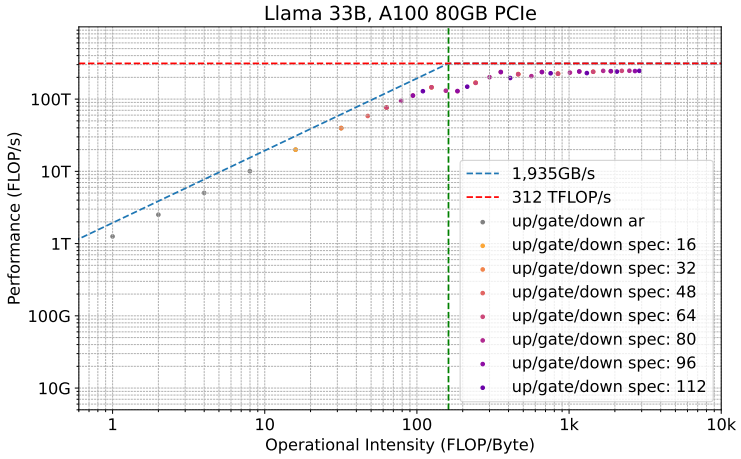


Figure 6.19: FLOP/s vs. operational intensity of Linear layers.

Table 6.4: TFLOP/s (first row) and Operational Intensity (second row) of attention matrix multiplication with batch size 16 for Llama 33B on an A100 80GB PCIe.

Seq. Length	Number of Candidate Tokens							
	1	16	32	48	64	80	96	112
128	0.54 0.98	7.87 12.8	14.73 21.33	19.78 27.43	25.25 32.0	28.63 35.56	32.58 38.4	36.57 40.73
256	0.75 0.99	11.2 13.47	21.29 23.27	28.69 30.72	36.59 36.57	41.2 41.29	45.99 45.18	52.33 48.43
512	1.02 0.99	14.69 13.84	27.47 24.38	37.35 32.68	47.09 39.38	52.24 44.91	59.55 49.55	66.35 53.49
1024	1.24 0.99	17.42 14.03	32.15 24.98	43.89 33.76	54.8 40.96	60.19 46.97	68.28 52.07	75.45 56.44
2048	1.39 0.99	19.03 14.12	35.05 25.28	48.03 34.32	59.66 41.8	63.91 48.08	72.83 53.43	80.05 58.04
4096	1.48 0.99	19.8 14.17	36.59 25.44	50.4 34.61	62.29 42.23	65.84 48.65	74.86 54.13	82.06 58.87
8192	1.53 0.99	20.08 14.2	36.89 25.52	50.44 34.76	62.11 42.45	67.5 48.94	76.97 54.49	84.5 59.3

Table 6.5: TFLOP/s (first row) and Operational Intensity (second row) of attention matrix multiplication with sequence length 1024 for Llama 33B on an A100 80GB PCIe.

Batch Size	Number of Candidate Tokens							
	1	16	32	48	64	80	96	112
1	0.37 0.99	5.22 14.03	10.15 24.98	15.02 33.76	19.79 40.96	21.52 46.97	25.65 52.07	29.4 56.44
2	0.54 0.99	8.25 14.03	16.0 24.98	21.62 33.76	28.24 40.96	31.84 46.97	37.49 52.07	43.04 56.44
4	0.75 0.99	11.41 14.03	21.97 24.98	30.02 33.76	38.71 40.96	43.41 46.97	50.06 52.07	56.77 56.44
8	1.02 0.99	14.78 14.03	27.78 24.98	38.09 33.76	47.99 40.96	53.32 46.97	61.0 52.07	68.11 56.44
16	1.24 0.99	17.42 14.03	32.15 24.98	43.89 33.76	54.8 40.96	60.19 46.97	68.28 52.07	75.45 56.44
32	1.39 0.99	18.89 14.03	34.67 24.98	47.57 33.76	58.89 40.96	63.61 46.97	72.17 52.07	79.21 56.44
64	1.48 0.99	19.58 14.03	35.87 24.98	49.45 33.76	61.13 40.96	64.84 46.97	73.73 52.07	81.02 56.44

6.4.3.3 Simulating Medusa Performance

We further employ a straightforward analytical model for the acceleration rate. The ablation study results in Sec. 6.4.2.4 indicate that the acceleration rate can be approximated by a simple logarithmic function. Using the results from Fig. 6.6a, we model the curve as $\texttt{acc\_rate} = 0.477 \log(\texttt{num\_candidate})$. We simulate the latency of one simplified block of the Llama-7B model (sequentially processing XW_Q, XW_K, XW_V, QK^T, PV, XW_u, XW_g, XW_d)

Table 6.6: TFLOP/s (first row) and Operational Intensity (second row) of linear layers (up/gate/down) for Llama 33B on an A100 80GB PCIe.

Batch Size	Number of Candidate Tokens							
	1	16	32	48	64	80	96	112
1	1.26 1.0	19.95 15.95	39.69 31.79	58.4 47.53	76.57 63.17	94.4 78.7	111.91 94.14	128.64 109.47
2	2.51 2.0	39.66 31.79	76.53 63.17	112.05 94.14	145.73 124.71	130.67 154.89	129.1 184.69	148.56 214.12
4	5.03 4.0	76.44 63.17	145.8 124.71	128.85 184.69	167.85 243.17	201.19 300.21	236.93 355.85	195.91 410.14
8	10.06 7.99	145.72 124.71	168.26 243.17	236.83 355.85	221.11 463.14	207.79 565.44	236.95 663.07	227.8 756.36
16	19.96 15.95	168.35 243.17	221.41 463.14	237.5 663.07	224.71 845.59	232.49 1012.87	241.12 1166.74	229.25 1308.76
32	39.69 31.79	221.74 463.14	224.88 845.59	241.33 1166.74	239.02 1440.25	245.83 1675.97	243.55 1881.24	240.33 2061.59
64	76.57 63.17	225.19 845.59	239.2 1440.25	243.26 1881.24	246.16 2221.31	246.91 2491.55	244.52 2711.46	246.14 2893.91

by first fixing the batch size at one and the sequence length at 1024. The candidate tokens are processed parallelly by constructing the tree attention described in Section 6.3.1.2. We omit the latency of the post-processing steps including verification and acceptance for Medusa since they introduce marginal overhead. Fig. 6.20 illustrates the simulated acceleration rate and speedup for different numbers of candidate tokens under these settings. As the number of candidate tokens increases, both the acceleration rate and speedup initially show improvements. However, beyond 64, the speedup starts to decline, indicating diminishing returns with further increases in candidate length. This aligns with the experimental results in Fig. 6.6b and suggests that there is an optimal range for the numbers of candidate tokens where Medusa provides the most significant performance gains.

We plot the simulated speedup under different batch size settings with a fixed sequence length of 1024 in Fig. 6.21. The results indicate that when the batch size exceeds 32, the speedup decreases and may even have a negative effect. This occurs because the linear layers shift from being memory-bandwidth-bound to computationally bound.

We conduct another experiment using a batch size of four and different sequence lengths. As shown in Fig. 6.22, the optimal number of candidate tokens remains relatively consistent across different sequence lengths. However, as the sequence length increases, the overall performance decreases. This performance drop is primarily due to the overhead from attention matrix multiplication, while the linear layer computation remains constant since

the computation of linear layers is independent of the sequence length.

Our simulations show that the optimal number of candidate tokens is key for model scaling with Medusa, as benefits decrease beyond a certain range. Initially, increasing batch size improves performance through parallelism, but too large a batch size shifts linear layers from memory-bandwidth-bound to compute-bound, reducing speedup. Longer sequences increase attention matrix multiplication overhead, lowering performance, and emphasizing the need to optimize attention mechanisms. Effective model scaling requires balancing the number of candidate tokens, adjusting batch sizes to avoid compute-bound transitions, and enhancing attention mechanisms for longer sequences. These strategies ensure better resource utilization and higher performance, demonstrating the value of simulations in predicting performance and guiding acceleration strategy design.

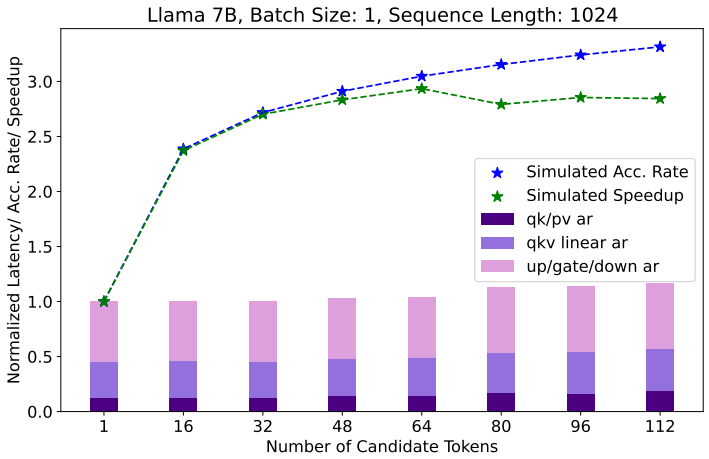


Figure 6.20: Simulated acceleration rate, speedup, and normalized latency ablation using different query lengths under the setting of batch size 1 and sequence length 1024 for Llama-7B.

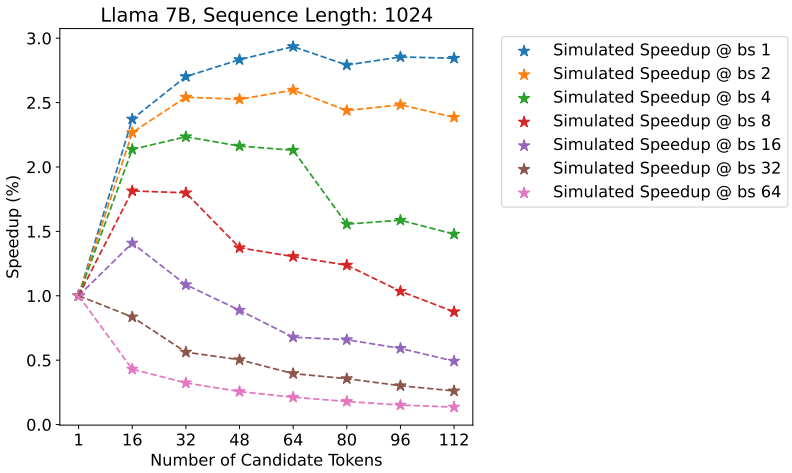


Figure 6.21: Simulated speedup with sequence length 1024 for Llama-7B.

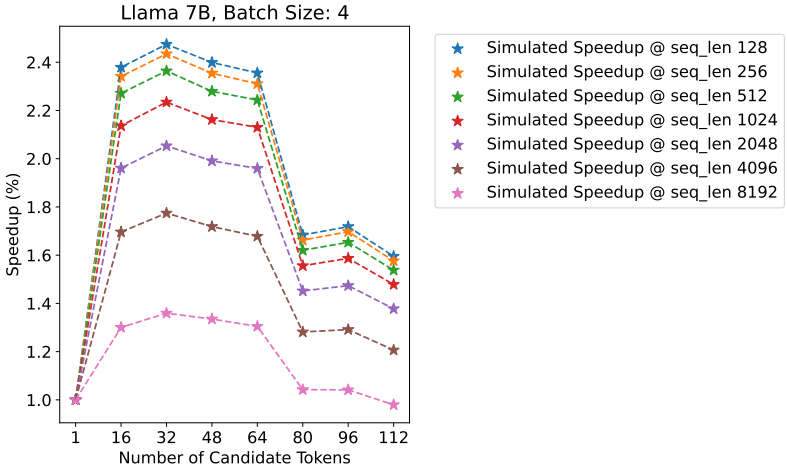


Figure 6.22: Simulated speedup with batch size 4 for Llama-7B.

6.5 Summary

In conclusion, Medusa enhances LLM inference speed by 2.3-2.8 times by equipping models with additional predictive decoding heads, allowing for the generation of multiple tokens simultaneously and bypassing the sequential decoding limitation. Key advantages of Medusa include its simplicity, parameter efficiency, and ease of integration into existing systems. Medusa avoids the need for specialized draft models. The typical acceptance scheme removes complications from rejection sampling while still providing reasonable outputs. Our approach including two efficient training procedures, ensures high-quality output across various models and prompt types. Also, Medusa introduces novel automated extensions, such as a searched sparse tree attention mechanism and a self-distillation pipeline, which further enhance its performance and adaptability. Moreover, we explored the hardware constraints, specifically memory-bandwidth bound, and their impact on Medusa. By profiling the performance of FLOP/s vs. Operational Intensity across various GPUs, we examined the changes when using Medusa for different operators. We propose a straightforward analytical model that allows us to predict the acceleration rates of Medusa, providing insights into the effects under different model sizes, sequence lengths, and batch sizes on target devices. Medusa improves the compute resource utilization ratio, offering solutions that not only improve model runtime but also enhance hardware performance and efficiency. The combination of these automated extensions, along with Medusa's core architecture and training procedures, creates a powerful and flexible framework for accelerating LLM inference. By providing a streamlined pipeline for quickly adapting Medusa to new models, this approach contributes to the decentralization of the LLM community, making it easier for researchers and practitioners to develop and deploy efficient LLMs tailored to their specific needs. As a result, Medusa has the potential to democratize access to high-performance LLMs and foster innovation across a wide range of natural language processing applications.

CHAPTER 7

CONCLUSION

7.1 Summary of Contributions

This dissertation investigates the challenges and opportunities in neural architecture design, focusing on efficient search and optimization methods to promote the development of innovative, high-performance neural networks for various applications. The research has made contributions to the field of deep learning by addressing key challenges and expanding the understanding of the factors that influence the performance of neural architectures. The primary contributions are as follows:

- The development of the Efficient Differentiable DNN (EDD) methodology significantly reduces search time and increases adaptability to various devices by simultaneously optimizing AI algorithms and hardware implementations.

- The proposal of a novel and generic NAS framework, GenNAS, which does not rely on task-specific labels for architecture evaluation and demonstrates remarkable efficiency in evaluating neural architectures and convergence speed for training.

- The introduction of Eproxy, a sophisticated proxy-based approach that leverages self-supervised learning and few-shot techniques to drastically reduce the computational expenses of neural architecture search. The Discrete Proxy Search (DPS) method is proposed to find optimized training settings for Eproxy, making it easily extensible to search spaces in the wild.

- The finding of critical principles that contribute to effective global convolutional models, leading to the proposal of the Structured Global

Convolution (SGConv) model. The efficient parameterization for the global convolution kernel in SGConv is further optimized through automated techniques such as kernel pruning and architecture search, enabling the discovery of more efficient and adaptable architectures for a wide range of tasks.

- The introduction of Medusa, a generic adaptor for large language models (LLMs) that addresses the memory-bandwidth bound nature of LLMs decoding phase through an automated framework for training and deploying adaptors, enabling practical applications of search and optimization strategies in real-world AI systems. Our hardware study further underscores Medusa's impact by showing how it improves compute resource utilization, enhancing both model runtime and hardware efficiency.

7.2 Implications and Future Directions

The research conducted in this dissertation has several implications for the field of neural architecture design in deep learning. The insights gained from this research can be used to guide the design of future neural architectures and optimization algorithms, ultimately contributing to the growing body of knowledge in deep learning. Several future directions are explored:

- Extend the proposed search methodologies to handle a wide range of emerging neural network architectures, such as state-space models, mixture of experts and hybrid large models.

- Investigate integrating the proposed NAS methods with other automated machine learning (AutoML) aspects, such as data preprocessing, feature engineering, and model selection, to create a fully automated end-to-end pipeline for deep learning applications.

- Further develop and optimize the Eproxy and DPS methods to handle even larger-scale search spaces and more diverse datasets, further reducing the computational resources required for NAS and making the process more accessible to researchers and practitioners with limited resources.

- Explore the potential of integrating SGConv with other architectural components and search strategies to create more powerful and efficient hybrid architectures for a wide range of tasks across different domains.
- Investigate the scalability and adaptability of Medusa to other types of foundation models and explore the potential of integrating Medusa with other optimization techniques to further improve the efficiency and performance of large-scale AI systems.

7.3 Final Remarks

In conclusion, this dissertation has made significant contributions to the neural architecture design field by addressing the challenges of efficiency, scalability, proxy evaluation, and interpretability. Through developing novel search methodologies, efficient proxies, and the investigation of complex neural architectures and their design principles, this research has advanced the state-of-the-art automation and optimization methods and contributed to developing more effective and adaptable deep learning solutions for various tasks and domains. The insights gained from this research will provide a solid foundation for future work in the field of neural architecture design and optimization, paving the way for more efficient, scalable, and interpretable AI systems that can tackle the ever-growing complexity and diversity of real-world problems.

REFERENCES

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

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

[3] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly et al., "An image is worth 16x16 words: Transformers for image recognition at scale," *arXiv preprint arXiv:2010.11929*, 2020.

[4] J. Ho, A. Jain, and P. Abbeel, "Denoising diffusion probabilistic models," *Advances in neural information processing systems*, vol. 33, pp. 6840–6851, 2020.

[5] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," *arXiv preprint arXiv:1706.03762*, 2017.

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

[7] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[8] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell et al., "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[9] G. Hinton, L. Deng, D. Yu, G. Dahl, A.-r. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, B. Kingsbury et al., "Deep neural networks for acoustic modeling in speech recognition," *IEEE Signal processing magazine*, vol. 29, 2012.

[10] A. Radford, J. W. Kim, T. Xu, G. Brockman, C. McLeavey, and I. Sutskever, "Robust speech recognition via large-scale weak supervision," in *International Conference on Machine Learning*. PMLR, 2023, pp. 28 492–28 518.

[11] Q. Wen, T. Zhou, C. Zhang, W. Chen, Z. Ma, J. Yan, and L. Sun, "Transformers in time series: A survey," *arXiv preprint arXiv:2202.07125*, 2022.

[12] H. Zhou, S. Zhang, J. Peng, S. Zhang, J. Li, H. Xiong, and W. Zhang, "Informer: Beyond efficient transformer for long sequence time-series forecasting," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 35, no. 12, 2021, pp. 11 106–11 115.

[13] A. Katharopoulos, A. Vyas, N. Pappas, and F. Fleuret, "Transformers are rnns: Fast autoregressive transformers with linear attention," in *International conference on machine learning*. PMLR, 2020, pp. 5156–5165.

[14] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," *arXiv preprint arXiv:1609.02907*, 2016.

[15] W. Hamilton, Z. Ying, and J. Leskovec, "Inductive representation learning on large graphs," *Advances in neural information processing systems*, vol. 30, 2017.

[16] A. Grover and J. Leskovec, "node2vec: Scalable feature learning for networks," in *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*, 2016, pp. 855–864.

[17] B. Zoph and Q. V. Le, "Neural architecture search with reinforcement learning," in *Proc. of ICLR*, 2017.

[18] H. Liu et al., "Darts: Differentiable architecture search," *arXiv:1806.09055*, 2018.

[19] M. S. Abdelfattah, A. Mehrotra, Ł. Dudziak, and N. D. Lane, "Zero-cost proxies for lightweight nas," *arXiv preprint arXiv:2101.08134*, 2021.

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

[21] S. Wang, B. Z. Li, M. Khabsa, H. Fang, and H. Ma, "Linformer: Self-attention with linear complexity," 2020.

[22] N. Kitaev, L. Kaiser, and A. Levskaya, "Reformer: The efficient transformer," in *International Conference on Learning Representations*, 2020. [Online]. Available: https://openreview.net/forum?id=rkgNKkHtvB

[23] Q. Chen, Q. Wu, J. Wang, Q. Hu, T. Hu, E. Ding, J. Cheng, and J. Wang, "Mixformer: Mixing features across windows and dimensions," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2022, pp. 5249–5259.

[24] T. N. Sainath, O. Vinyals, A. Senior, and H. Sak, "Convolutional, long short-term memory, fully connected deep neural networks," in *2015 IEEE international conference on acoustics, speech and signal processing (ICASSP)*. Ieee, 2015, pp. 4580–4584.

[25] W. Fedus, B. Zoph, and N. Shazeer, "Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity," *Journal of Machine Learning Research*, vol. 23, no. 120, pp. 1–39, 2022.

[26] N. Shazeer, A. Mirhoseini, K. Maziarz, A. Davis, Q. Le, G. Hinton, and J. Dean, "Outrageously large neural networks: The sparsely-gated mixture-of-experts layer," *arXiv preprint arXiv:1701.06538*, 2017.

[27] H. Cai et al., "Proxylessnas: Direct neural architecture search on target task and hardware," *arXiv:1812.00332*, 2018.

[28] X. Zhang et al., "DNNBuilder: an automated tool for building high-performance DNN hardware accelerators for FPGAs," in *ICCAD*, 2018.

[29] B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le, "Learning transferable architectures for scalable image recognition," in *Proc. of CVPR*, 2018.

[30] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," in *Proc. of CVPR*, 2017.

[31] E. Real, A. Aggarwal, Y. Huang, and Q. V. Le, "Regularized evolution for image classifier architecture search," in *Proc. of AAAI*, 2019.

[32] Y. Chen et al., "Cloud-dnn: An open framework for mapping dnn models to cloud fpgas," in *FPGA*, 2019.

[33] C. Hao et al., "FPGA/DNN Co-Design: An Efficient Design Methodology for IoT Intelligence on the Edge," *DAC*, 2019.

[34] M. Tan et al., "Mnasnet: Platform-aware neural architecture search for mobile," in *CVPR*, 2019.

[35] D. Stamoulis et al., "Single-path NAS: Designing hardware-efficient convnets in less than 4 hours," *arXiv:1904.02877*, 2019.

[36] B. Wu et al., "Fbnet: Hardware-aware efficient convnet design via differentiable neural architecture search," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2019, pp. 10 734–10 742.

[37] S. V. K. Srinivas et al., "Hardware Aware Neural Network Architectures using FbNet," *arXiv:1906.07214*, 2019.

[38] C. Gong et al., "Mixed Precision Neural Architecture Search for Energy Efficient Deep Learning," 2019.

[39] W. Jiang et al., "Accuracy vs. Efficiency: Achieving Both through FPGA-Implementation Aware Neural Architecture Search," in *DAC*. ACM, 2019.

[40] C. Hao et al., "NAIS: Neural Architecture and Implementation Search and its Applications in Autonomous Driving," 2019.

[41] K. Wang et al., " HAQ: Hardware-Aware Automated Quantization with Mixed Precision," in *CVPR*, 2019.

[42] K. Guo et al., "A Survey of FPGA-based Neural Network Inference Accelerators," *ACM TRETS*, vol. 12, no. 1, p. 2, 2019.

[43] E. Polak, *Optimization: algorithms and consistent approximations*. Springer Science & Business Media, 2012, vol. 124.

[44] P. Judd et al., "Stripes: Bit-serial deep neural network computing," in *MICRO*. IEEE, 2016.

[45] S. Sharify et al., "Loom: Exploiting weight and activation precisions to accelerate convolutional neural networks," in *DAC*. IEEE, 2018.

[46] H. Sharma et al., "Bit fusion: Bit-level dynamically composable architecture for accelerating deep neural networks," in *ISCA*. IEEE, 2018.

[47] "CHaiDNN," https://github.com/Xilinx/CHaiDNN, accessed: 2019-11-27.

[48] M. Sandler et al., "Mobilenetv2: Inverted residuals and linear bottlenecks," in *CVPR*, 2018.

[49] N. Ma et al., "Shufflenet v2: Practical guidelines for efficient cnn architecture design," in *ECCV*, 2018.

[50] X. Su, S. You, J. Xie, M. Zheng, F. Wang, C. Qian, C. Zhang, X. Wang, and C. Xu, "Vision transformer architecture search," *arXiv preprint arXiv:2106.13700*, 2021.

[51] C. Liu, L.-C. Chen, F. Schroff, H. Adam, W. Hua, A. L. Yuille, and L. Fei-Fei, "Auto-deeplab: Hierarchical neural architecture search for semantic image segmentation," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2019, pp. 82–92.

[52] V. Nekrasov, H. Chen, C. Shen, and I. Reid, "Fast neural architecture search of compact semantic segmentation models via auxiliary cells," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2019, pp. 9126–9135.

[53] A. Shaw, D. Hunter, F. Landola, and S. Sidhu, "Squeezenas: Fast neural architecture search for faster semantic segmentation," in *Proceedings of the IEEE/CVF International Conference on Computer Vision Workshops*, 2019, pp. 0–0.

[54] C. Li, T. Tang, G. Wang, J. Peng, B. Wang, X. Liang, and X. Chang, "Bossnas: Exploring hybrid cnn-transformers with block-wisely self-supervised neural architecture search," *arXiv preprint arXiv:2103.12424*, 2021.

[55] H. Wang, Z. Wu, Z. Liu, H. Cai, L. Zhu, C. Gan, and S. Han, "Hat: Hardware-aware transformers for efficient natural language processing," *arXiv preprint arXiv:2005.14187*, 2020.

[56] D. So, Q. Le, and C. Liang, "The evolved transformer," in *International Conference on Machine Learning*. PMLR, 2019, pp. 5877–5886.

[57] C. Hao and D. Chen, "Deep neural network model and fpga accelerator co-design: Opportunities and challenges," in *2018 14th IEEE International Conference on Solid-State and Integrated Circuit Technology (ICSICT)*. IEEE, 2018, pp. 1–4.

[58] X. Zhang, H. Lu, C. Hao, J. Li, B. Cheng, Y. Li, K. Rupnow, J. Xiong, T. Huang, H. Shi et al., "Skynet: a hardware-efficient method for object detection and tracking on embedded systems," *arXiv preprint arXiv:1909.09709*, 2019.

[59] J. Lin, W.-M. Chen, Y. Lin, J. Cohn, C. Gan, and S. Han, "Mcunet: Tiny deep learning on iot devices," *arXiv preprint arXiv:2007.10319*, 2020.

[60] J. Lin, W.-M. Chen, H. Cai, C. Gan, and S. Han, "Mcunetv2: Memory-efficient patch-based inference for tiny deep learning," *arXiv preprint arXiv:2110.15352*, 2021.

[61] E. Real, S. Moore, A. Selle, S. Saxena, Y. L. Suematsu, J. Tan, Q. V. Le, and A. Kurakin, "Large-scale evolution of image classifiers," in *Proc of ICML*, 2017.

[62] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. of CVPR*, 2016.

[63] L.-C. Chen, Y. Zhu, G. Papandreou, F. Schroff, and H. Adam, "Encoder-decoder with atrous separable convolution for semantic image segmentation," in *Proceedings of the European conference on computer vision (ECCV)*, 2018, pp. 801–818.

[64] M. Marsden, K. McGuinness, S. Little, and N. E. O'Connor, "Resnetcrowd: A residual deep learning architecture for crowd counting, violent behaviour detection and crowd density level classification," in *2017 14th IEEE International Conference on Advanced Video and Signal Based Surveillance (AVSS)*. IEEE, 2017, pp. 1–7.

[65] M. Raghu, C. Zhang, J. Kleinberg, and S. Bengio, "Transfusion: Understanding transfer learning for medical imaging," *arXiv preprint arXiv:1902.07208*, 2019.

[66] M. A. Hearst, S. T. Dumais, E. Osuna, J. Platt, and B. Scholkopf, "Support vector machines," *IEEE Intelligent Systems and their applications*, vol. 13, no. 4, pp. 18–28, 1998.

[67] A. Jacot, F. Gabriel, and C. Hongler, "Neural tangent kernel: Convergence and generalization in neural networks," *arXiv preprint arXiv:1806.07572*, 2018.

[68] W. W. Daniel et al., "Applied nonparametric statistics," 1990.

[69] D. Zhou, X. Zhou, W. Zhang, C. C. Loy, S. Yi, X. Zhang, and W. Ouyang, "Econas: Finding proxies for economical neural architecture search," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2020, pp. 11 396–11 404.

[70] X. Chu, B. Zhang, R. Xu, and J. Li, "Fairnas: Rethinking evaluation fairness of weight sharing neural architecture search," *arXiv preprint arXiv:1907.01845*, 2019.

[71] L. Li and A. Talwalkar, "Random search and reproducibility for neural architecture search," *arXiv preprint arXiv:1902.07638*, 2019.

[72] N. Klyuchnikov, I. Trofimov, E. Artemova, M. Salnikov, M. Fedorov, and E. Burnaev, "Nas-bench-nlp: neural architecture search benchmark for natural language processing," *arXiv preprint arXiv:2006.07116*, 2020.

[73] C. Ying, A. Klein, E. Christiansen, E. Real, K. Murphy, and F. Hutter, "Nas-bench-101: Towards reproducible neural architecture search," in *International Conference on Machine Learning*. PMLR, 2019, pp. 7105–7114.

[74] Y. Xu, L. Xie, X. Zhang, X. Chen, G.-J. Qi, Q. Tian, and H. Xiong, "Pc-darts: Partial channel connections for memory-efficient architecture search," *arXiv preprint arXiv:1907.05737*, 2019.

[75] Y. Li, C. Hao, X. Zhang, X. Liu, Y. Chen, J. Xiong, W.-m. Hwu, and D. Chen, "Edd: Efficient differentiable dnn architecture and implementation co-search for embedded ai solutions," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2020, pp. 1–6.

[76] A. Zela, T. Elsken, T. Saikia, Y. Marrakchi, T. Brox, and F. Hutter, "Understanding and robustifying differentiable architecture search," *arXiv preprint arXiv:1909.09656*, 2019.

[77] X. Dong and Y. Yang, "Nas-bench-201: Extending the scope of reproducible neural architecture search," *arXiv preprint arXiv:2001.00326*, 2020.

[78] K. Yu, C. Sciuto, M. Jaggi, C. Musat, and M. Salzmann, "Evaluating the search phase of neural architecture search," *arXiv preprint arXiv:1902.08142*, 2019.

[79] J. Mellor, J. Turner, A. Storkey, and E. J. Crowley, "Neural architecture search without training," *arXiv preprint arXiv:2006.04647*, 2020.

[80] D. Dey, S. Shah, and S. Bubeck, "Fear: A simple lightweight method to rank architectures," *arXiv preprint arXiv:2106.04010*, 2021.

[81] G. Li, S. K. Mandal, U. Y. Ogras, and R. Marculescu, "Flash: Fast neural architecture search with hardware optimization," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 20, no. 5s, pp. 1–26, 2021.

[82] J. Siems, L. Zimmer, A. Zela, J. Lukasik, M. Keuper, and F. Hutter, "Nas-bench-301 and the case for surrogate benchmarks for neural architecture search," *arXiv preprint arXiv:2008.09777*, 2020.

[83] X. Su, T. Huang, Y. Li, S. You, F. Wang, C. Qian, C. Zhang, and C. Xu, "Prioritized architecture sampling with monto-carlo tree search," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2021, pp. 10 968–10 977.

[84] S. Kornblith, J. Shlens, and Q. V. Le, "Do better imagenet models transfer better?" in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2019, pp. 2661–2671.

[85] C. Liu, P. Dollár, K. He, R. Girshick, A. Yuille, and S. Xie, "Are labels necessary for neural architecture search?" in *European Conference on Computer Vision*. Springer, 2020, pp. 798–813.

[86] L. Jing and Y. Tian, "Self-supervised visual feature learning with deep neural networks: A survey," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2020.

[87] A. Dosovitskiy, J. T. Springenberg, M. Riedmiller, and T. Brox, "Discriminative unsupervised feature learning with convolutional neural networks." Citeseer, 2014.

[88] C. Doersch, A. Gupta, and A. A. Efros, "Unsupervised visual representation learning by context prediction," in *Proceedings of the IEEE international conference on computer vision*, 2015, pp. 1422–1430.

[89] X. Wang and A. Gupta, "Unsupervised learning of visual representations using videos," in *Proceedings of the IEEE international conference on computer vision*, 2015, pp. 2794–2802.

[90] H. Wang, X. Wu, Z. Huang, and E. P. Xing, "High-frequency component helps explain the generalization of convolutional neural networks," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2020, pp. 8684–8694.

[91] J. Long, E. Shelhamer, and T. Darrell, "Fully convolutional networks for semantic segmentation," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 3431–3440.

[92] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[93] T. Mikolov, M. Karafiát, L. Burget, J. Černockỳ, and S. Khudanpur, "Recurrent neural network based language model," in *Eleventh annual conference of the international speech communication association*, 2010.

[94] Z.-Q. J. Xu, Y. Zhang, T. Luo, Y. Xiao, and Z. Ma, "Frequency principle: Fourier analysis sheds light on deep neural networks," *arXiv preprint arXiv:1901.06523*, 2019.

[95] G. P. Tolstov, *Fourier series*. Courier Corporation, 2012.

[96] S. J. Montgomery-Smith, "The distribution of rademacher sums," *Proceedings of the American Mathematical Society*, vol. 109, no. 2, pp. 517–522, 1990.

[97] I. Radosavovic, J. Johnson, S. Xie, W.-Y. Lo, and P. Dollár, "On network design spaces for visual recognition," in *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2019, pp. 1882–1890.

[98] M. Marcus, B. Santorini, and M. A. Marcinkiewicz, "Building a large annotated corpus of english: The penn treebank," 1993.

[99] X. Dong and Y. Yang, "Searching for a robust neural architecture in four gpu hours," in *Proc. of CVPR*, 2019, pp. 1761–1770.

[100] X. Dong and Y. Yang, "One-shot neural architecture search via self-evaluated template network," in *Proceedings of the IEEE International Conference on Computer Vision*, 2019, pp. 3681–3690.

[101] H. Pham, M. Y. Guan, B. Zoph, Q. V. Le, and J. Dean, "Efficient neural architecture search via parameter sharing," in *Proc. of ICML*, 2018.

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

[103] W. Chen, X. Gong, and Z. Wang, "Neural architecture search on imagenet in four gpu hours: A theoretically inspired perspective," *arXiv preprint arXiv:2102.11535*, 2021.

[104] C. Liu, B. Zoph, M. Neumann, J. Shlens, W. Hua, L.-J. Li, L. Fei-Fei, A. Yuille, J. Huang, and K. Murphy, "Progressive neural architecture search," in *Proc. of ECCV*, 2018, pp. 19–34.

[105] S. Xie, H. Zheng, C. Liu, and L. Lin, "Snas: stochastic neural architecture search," *Proc. of ICLR*, 2019.

[106] X. Chen, L. Xie, J. Wu, and Q. Tian, "Progressive differentiable architecture search: Bridging the depth gap between search and evaluation," in *Proceedings of the IEEE International Conference on Computer Vision*, 2019, pp. 1294–1303.

[107] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 779–788.

[108] M. Schuster and K. K. Paliwal, "Bidirectional recurrent neural networks," *IEEE transactions on Signal Processing*, vol. 45, no. 11, pp. 2673–2681, 1997.

[109] Z. Wu, Z. Liu, J. Lin, Y. Lin, and S. Han, "Lite transformer with long-short range attention," *arXiv preprint arXiv:2004.11886*, 2020.

[110] H. Cai, C. Gan, T. Wang, Z. Zhang, and S. Han, "Once-for-all: Train one network and specialize it for efficient deployment," *arXiv preprint arXiv:1908.09791*, 2019.

[111] Y. Weng, T. Zhou, Y. Li, and X. Qiu, "Nas-unet: Neural architecture search for medical image segmentation," *IEEE Access*, vol. 7, pp. 44 247–44 257, 2019.

[112] N. Wang, Y. Gao, H. Chen, P. Wang, Z. Tian, C. Shen, and Y. Zhang, "Nas-fcos: Fast neural architecture search for object detection," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2020, pp. 11 943–11 951.

[113] Z. Liu, H. Tang, S. Zhao, K. Shao, and S. Han, "Pvnas: 3d neural architecture search with point-voxel convolution," *arXiv preprint arXiv:2204.11797*, 2022.

[114] X. Gong, S. Chang, Y. Jiang, and Z. Wang, "Autogan: Neural architecture search for generative adversarial networks," in *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2019, pp. 3224–3234.

[115] J. Peng, J. Zhang, C. Li, G. Wang, X. Liang, and L. Lin, "Pi-nas: Improving neural architecture search by reducing supernet training consistency shift," in *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2021, pp. 12 354–12 364.

[116] X. Zhang, P. Hou, X. Zhang, and J. Sun, "Neural architecture search with random labels," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2021, pp. 10 907–10 916.

[117] M. Zhang, S. Su, S. Pan, X. Chang, W. Huang, B. Yang, and G. Haffari, "Differentiable architecture search meets network pruning at initialization: A more reliable, efficient, and flexible framework," *arXiv preprint arXiv:2106.11542*, 2021.

[118] M. Lin, P. Wang, Z. Sun, H. Chen, X. Sun, Q. Qian, H. Li, and R. Jin, "Zen-nas: A zero-shot nas for high-performance image recognition," in *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2021, pp. 347–356.

[119] G. Li, Y. Yang, K. Bhardwaj, and R. Marculescu, "Zico: Zero-shot nas via inverse coefficient of variation on gradients," *arXiv preprint arXiv:2301.11300*, 2023.

[120] X. Dong and Y. Yang, "Nas-bench-201: Extending the scope of reproducible neural architecture search," in *International Conference on Learning Representations (ICLR)*, 2020. [Online]. Available: https://openreview.net/forum?id=HJxyZkBKDr

[121] M. Ding, Y. Huo, H. Lu, L. Yang, Z. Wang, Z. Lu, J. Wang, and P. Luo, "Learning versatile neural architectures by propagating network codes," *arXiv preprint arXiv:2103.13253*, 2021.

[122] H. Chen, M. Lin, X. Sun, and H. Li, "Nas-bench-zero: A large scale dataset for understanding zero-shot neural architecture search," 2021.

[123] X. Ning, C. Tang, W. Li, Z. Zhou, S. Liang, H. Yang, and Y. Wang, "Evaluating efficient performance estimators of neural architectures," *Advances in Neural Information Processing Systems*, vol. 34, 2021.

[124] Y. Li, C. Hao, P. Li, J. Xiong, and D. Chen, "Generic neural architecture search via regression," *Advances in Neural Information Processing Systems*, vol. 34, pp. 20 476–20 490, 2021.

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

[126] R. Girshick, "Fast r-cnn," in *Proceedings of the IEEE international conference on computer vision*, 2015, pp. 1440–1448.

[127] L. Bertinetto, J. Valmadre, J. F. Henriques, A. Vedaldi, and P. H. Torr, "Fully-convolutional siamese networks for object tracking," in *European conference on computer vision*. Springer, 2016, pp. 850–865.

[128] B. Li, J. Yan, W. Wu, Z. Zhu, and X. Hu, "High performance visual tracking with siamese region proposal network," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 8971–8980.

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

[130] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *Proceedings of the thirteenth international conference on artificial intelligence and statistics*. JMLR Workshop and Conference Proceedings, 2010, pp. 249–256.

[131] G. Alain and Y. Bengio, "Understanding intermediate layers using linear classifier probes," *arXiv preprint arXiv:1610.01644*, 2016.

[132] M. Javaheripi, S. Shah, S. Mukherjee, T. L. Religa, C. C. Mendes, G. H. de Rosa, S. Bubeck, F. Koushanfar, and D. Dey, "Litetransformersearch: Training-free on-device search for efficient autoregressive language models," *arXiv preprint arXiv:2203.02094*, 2022.

[133] C. White, M. Khodak, R. Tu, S. Shah, S. Bubeck, and D. Dey, "A deeper look at zero-cost proxies for lightweight nas," in *ICLR Blog Track*, 2022, https://iclr-blog-track.github.io/2022/03/25/zero-cost-proxies/. [Online]. Available: https://iclr-blog-track.github.io/2022/03/25/zero-cost-proxies/

[134] D. Dey, S. Shah, and S. Bubeck, "Ranking architectures by feature extraction capabilities," in *8th ICML Workshop on Automated Machine Learning (AutoML)*, 2021.

[135] I. Radosavovic, R. P. Kosaraju, R. Girshick, K. He, and P. Dollár, "Designing network design spaces," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2020, pp. 10 428–10 436.

[136] Y. Duan, X. Chen, H. Xu, Z. Chen, X. Liang, T. Zhang, and Z. Li, "Transnas-bench-101: Improving transferability and generalizability of cross-task neural architecture search," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2021, pp. 5251–5260.

[137] H. Li, Z. Xu, G. Taylor, C. Studer, and T. Goldstein, "Visualizing the loss landscape of neural nets," *Advances in neural information processing systems*, vol. 31, 2018.

[138] A. R. Zamir, A. Sax, W. Shen, L. J. Guibas, J. Malik, and S. Savarese, "Taskonomy: Disentangling task transfer learning," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 3712–3722.

[139] S. Xie, H. Zheng, C. Liu, and L. Lin, "Snas: stochastic neural architecture search," *arXiv preprint arXiv:1812.09926*, 2018.

[140] M. Cordts, M. Omran, S. Ramos, T. Rehfeld, M. Enzweiler, R. Benenson, U. Franke, S. Roth, and B. Schiele, "The cityscapes dataset for semantic urban scene understanding," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 3213–3223.

[141] A. Geiger, P. Lenz, and R. Urtasun, "Are we ready for autonomous driving? the kitti vision benchmark suite," in *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2012.

[142] H. Kuehne, H. Jhuang, E. Garrote, T. Poggio, and T. Serre, "Hmdb: a large video database for human motion recognition," in *2011 International conference on computer vision*. IEEE, 2011, pp. 2556–2563.

[143] M. Ding, Y. Huo, H. Lu, L. Yang, Z. Wang, Z. Lu, J. Wang, and P. Luo, "Learning Versatile Neural Architectures by Propagating Network Codes," https://github.com/dingmyu/NCP, 2023.

[144] R. Luo, F. Tian, T. Qin, E. Chen, and T.-Y. Liu, "Neural architecture optimization," *Advances in neural information processing systems*, vol. 31, 2018.

[145] R. Luo, X. Tan, R. Wang, T. Qin, E. Chen, and T.-Y. Liu, "Semi-supervised neural architecture search," *Advances in Neural Information Processing Systems*, vol. 33, pp. 10 547–10 557, 2020.

[146] J. Wu, X. Dai, D. Chen, Y. Chen, M. Liu, Y. Yu, Z. Wang, Z. Liu, M. Chen, and L. Yuan, "Stronger nas with weaker predictors," *Advances in Neural Information Processing Systems*, vol. 34, pp. 28 904–28 918, 2021.

[147] L. Wang, Y. Zhao, Y. Jinnai, Y. Tian, and R. Fonseca, "Alphax: exploring neural architectures with deep neural networks and monte carlo tree search," *arXiv preprint arXiv:1903.11059*, 2019.

[148] L. Wang, S. Xie, T. Li, R. Fonseca, and Y. Tian, "Sample-efficient neural architecture search by learning actions for monte carlo tree search," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2021.

[149] Y. Tay, M. Dehghani, S. Abnar, Y. Shen, D. Bahri, P. Pham, J. Rao, L. Yang, S. Ruder, and D. Metzler, "Long range arena: A benchmark for efficient transformers," *arXiv preprint arXiv:2011.04006*, 2020.

[150] A. Gu, K. Goel, and C. Ré, "Efficiently modeling long sequences with structured state spaces," *arXiv preprint arXiv:2111.00396*, 2021.

[151] D. Linsley, J. Kim, V. Veerabadran, C. Windolf, and T. Serre, "Learning long-range spatial dependencies with horizontal gated recurrent units," *Advances in neural information processing systems*, vol. 31, 2018.

[152] J. Kim, D. Linsley, K. Thakkar, and T. Serre, "Disentangling neural mechanisms for perceptual grouping," in *International Conference on Learning Representations*, 2019.

[153] A. Gu, T. Dao, S. Ermon, A. Rudra, and C. Ré, "Hippo: Recurrent memory with optimal polynomial projections," *Advances in Neural Information Processing Systems*, vol. 33, pp. 1474–1487, 2020.

[154] Z. Liu, Y. Lin, Y. Cao, H. Hu, Y. Wei, Z. Zhang, S. Lin, and B. Guo, "Swin transformer: Hierarchical vision transformer using shifted windows," in *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2021, pp. 10 012–10 022.

[155] L. Dong, S. Xu, and B. Xu, "Speech-transformer: a no-recurrence sequence-to-sequence model for speech recognition," in *2018 IEEE international conference on acoustics, speech and signal processing (ICASSP)*. IEEE, 2018, pp. 5884–5888.

[156] W. Ye, H. Yang, S. Zhao, H. Fang, X. Shi, and N. Neppalli, "A transformer-based substitute recommendation model incorporating weakly supervised customer behavior data," *arXiv preprint arXiv:2211.02533*, 2022.

[157] R. Child, S. Gray, A. Radford, and I. Sutskever, "Generating long sequences with sparse transformers," *arXiv preprint arXiv:1904.10509*, 2019.

[158] N. Kitaev, L. Kaiser, and A. Levskaya, "Reformer: The efficient transformer," in *International Conference on Learning Representations*, 2019.

[159] M. Zaheer, G. Guruganesh, K. A. Dubey, J. Ainslie, C. Alberti, S. Ontanon, P. Pham, A. Ravula, Q. Wang, L. Yang et al., "Big bird: Transformers for longer sequences," *Advances in Neural Information Processing Systems*, vol. 33, pp. 17 283–17 297, 2020.

[160] Y. Tay, D. Bahri, D. Metzler, D. Juan, Z. Zhao, and C. Zheng, "Synthesizer: Rethinking self-attention in transformer models. arxiv 2020," *arXiv preprint arXiv:2005.00743*, vol. 2, 2020.

[161] H. Peng, N. Pappas, D. Yogatama, R. Schwartz, N. Smith, and L. Kong, "Random feature attention," in *International Conference on Learning Representations*, 2021. [Online]. Available: https://openreview.net/forum?id=QtTKTdVrFBB

[162] Z. Qin, W. Sun, H. Deng, D. Li, Y. Wei, B. Lv, J. Yan, L. Kong, and Y. Zhong, "cosformer: Rethinking softmax in attention," in *International Conference on Learning Representations*, 2021.

[163] U. Shaham, E. Segal, M. Ivgi, A. Efrat, O. Yoran, A. Haviv, A. Gupta, W. Xiong, M. Geva, J. Berant, and O. Levy, "Scrolls: Standardized comparison over long language sequences," 2022.

[164] T. Salimans and D. P. Kingma, "Weight normalization: A simple reparameterization to accelerate training of deep neural networks," *Advances in neural information processing systems*, vol. 29, 2016.

[165] S. Zagoruyko and N. Komodakis, "Diracnets: Training very deep neural networks without skip-connections," *arXiv preprint arXiv:1706.00388*, 2017.

[166] X. Ding, Y. Guo, G. Ding, and J. Han, "Acnet: Strengthening the kernel skeletons for powerful cnn via asymmetric convolution blocks," in *Proceedings of the IEEE/CVF international conference on computer vision*, 2019, pp. 1911–1920.

[167] S. Guo, J. M. Alvarez, and M. Salzmann, "Expandnets: Linear over-parameterization to train compact convolutional networks," *Advances in Neural Information Processing Systems*, vol. 33, pp. 1298–1310, 2020.

[168] X. Ding, X. Zhang, N. Ma, J. Han, G. Ding, and J. Sun, "Repvgg: Making vgg-style convnets great again," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2021, pp. 13 733–13 742.

[169] J. Cao, Y. Li, M. Sun, Y. Chen, D. Lischinski, D. Cohen-Or, B. Chen, and C. Tu, "Do-conv: Depthwise over-parameterized convolutional layer," *IEEE Transactions on Image Processing*, 2022.

[170] Y. Rao, W. Zhao, Z. Zhu, J. Lu, and J. Zhou, "Global filter networks for image classification," *Advances in Neural Information Processing Systems*, vol. 34, pp. 980–993, 2021.

[171] J. Guibas, M. Mardani, Z. Li, A. Tao, A. Anandkumar, and B. Catanzaro, "Efficient token mixing for transformers via adaptive fourier neural operators," in *International Conference on Learning Representations*, 2021.

[172] D. W. Romero, A. Kuzina, E. J. Bekkers, J. M. Tomczak, and M. Hoogendoorn, "Ckconv: Continuous kernel convolution for sequential data," in *International Conference on Learning Representations*, 2021.

[173] D. W. Romero, R.-J. Bruintjes, J. M. Tomczak, E. J. Bekkers, M. Hoogendoorn, and J. van Gemert, "Flexconv: Continuous kernel convolutions with differentiable kernel sizes," in *International Conference on Learning Representations*, 2021.

[174] D. W. Romero, D. M. Knigge, A. Gu, E. J. Bekkers, E. Gavves, J. M. Tomczak, and M. Hoogendoorn, "Towards a general purpose cnn for long range dependencies in nd," *arXiv preprint arXiv:2206.03398*, 2022.

[175] A. Gu, I. Johnson, K. Goel, K. Saab, T. Dao, A. Rudra, and C. Ré, "Combining recurrent, convolutional, and continuous-time models with linear state space layers," *Advances in neural information processing systems*, vol. 34, pp. 572–585, 2021.

[176] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin et al., "Tensorflow: Large-scale machine learning on heterogeneous distributed systems," *arXiv preprint arXiv:1603.04467*, 2016.

[177] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang, "Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems," *arXiv preprint arXiv:1512.01274*, 2015.

[178] L. Chen, *Deep Learning and Practice with MindSpore*. Springer Nature, 2021.

[179] Y. Ma, D. Yu, T. Wu, and H. Wang, "Paddlepaddle: An open-source deep learning platform from industrial practice," *Frontiers of Data and Domputing*, vol. 1, no. 1, pp. 105–115, 2019.

[180] A. Gu, A. Gupta, K. Goel, and C. Ré, "On the parameterization and initialization of diagonal state space models," *arXiv preprint arXiv:2206.11893*, 2022.

[181] J. T. Smith, A. Warrington, and S. W. Linderman, "Simplified state space layers for sequence modeling," *arXiv preprint arXiv:2208.04933*, 2022.

[182] R. Hasani, M. Lechner, T.-H. Wang, M. Chahine, A. Amini, and D. Rus, "Liquid structural state-space models," *arXiv preprint arXiv:2209.12951*, 2022.

[183] A. Van den Oord, N. Kalchbrenner, L. Espeholt, O. Vinyals, A. Graves et al., "Conditional image generation with pixelcnn decoders," *Advances in neural information processing systems*, vol. 29, 2016.

[184] A. v. d. Oord, S. Dieleman, H. Zen, K. Simonyan, O. Vinyals, A. Graves, N. Kalchbrenner, A. Senior, and K. Kavukcuoglu, "Wavenet: A generative model for raw audio," *arXiv preprint arXiv:1609.03499*, 2016.

[185] A. Gu, I. Johnson, A. Timalsina, A. Rudra, and C. Ré, "How to train your hippo: State space models with generalized orthogonal basis projections," *arXiv preprint arXiv:2206.12037*, 2022.

[186] P. Warden, "Speech commands: A dataset for limited-vocabulary speech recognition," *arXiv preprint arXiv:1804.03209*, 2018.

[187] Z. Liu, H. Mao, C.-Y. Wu, C. Feichtenhofer, T. Darrell, and S. Xie, "A convnet for the 2020s," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2022, pp. 11 976–11 986.

[188] P. Kidger, J. Morrill, J. Foster, and T. Lyons, "Neural controlled differential equations for irregular time series," *Advances in Neural Information Processing Systems*, vol. 33, pp. 6696–6707, 2020.

[189] Z. Dai, Z. Yang, Y. Yang, W. W. Cohen, J. Carbonell, Q. V. Le, and R. Salakhutdinov, "Transformer-xl: Attentive language models beyond a fixed-length context," *arXiv preprint arXiv:1901.02860*, 2019.

[190] S. Merity, C. Xiong, J. Bradbury, and R. Socher, "Pointer sentinel mixture models," *arXiv preprint arXiv:1609.07843*, 2016.

[191] Y. Zhu, R. Kiros, R. Zemel, R. Salakhutdinov, R. Urtasun, A. Torralba, and S. Fidler, "Aligning books and movies: Towards story-like visual explanations by watching movies and reading books," in *Proceedings of the IEEE international conference on computer vision*, 2015, pp. 19–27.

[192] W. Foundation, "Wikimedia downloads." [Online]. Available: https://dumps.wikimedia.org

[193] A. Wang, A. Singh, J. Michael, F. Hill, O. Levy, and S. R. Bowman, "Glue: A multi-task benchmark and analysis platform for natural language understanding," 2019, in the Proceedings of ICLR.

[194] G. Ke, D. He, and T.-Y. Liu, "Rethinking positional encoding in language pre-training," in *International Conference on Learning Representations*, 2020.

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

[196] H. Touvron, M. Cord, M. Douze, F. Massa, A. Sablayrolles, and H. Jégou, "Training data-efficient image transformers & distillation through attention," in *International Conference on Machine Learning*. PMLR, 2021, pp. 10 347–10 357.

[197] H. Touvron, M. Cord, A. Sablayrolles, G. Synnaeve, and H. Jégou, "Going deeper with image transformers," in *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2021, pp. 32–42.

[198] W. Yu, M. Luo, P. Zhou, C. Si, Y. Zhou, X. Wang, J. Feng, and S. Yan, "Metaformer is actually what you need for vision," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2022, pp. 10 819–10 829.

[199] J. Johnson, B. Hariharan, L. Van Der Maaten, L. Fei-Fei, C. Lawrence Zitnick, and R. Girshick, "Clevr: A diagnostic dataset for compositional language and elementary visual reasoning," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 2901–2910.

[200] F. Zhu, Y. Zhu, X. Chang, and X. Liang, "Vision-language navigation with self-supervised auxiliary reasoning tasks," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2020, pp. 10 012–10 022.

[201] A. Gupta, "Diagonal state spaces are as effective as structured state spaces," *arXiv preprint arXiv:2203.14343*, 2022.

[202] A. Chowdhery, S. Narang, J. Devlin, M. Bosma, G. Mishra, A. Roberts, P. Barham, H. W. Chung, C. Sutton, S. Gehrmann et al., "Palm: Scaling language modeling with pathways," *arXiv preprint arXiv:2204.02311*, 2022.

[203] S. Zhang, S. Roller, N. Goyal, M. Artetxe, M. Chen, S. Chen, C. Dewan, M. Diab, X. Li, X. V. Lin et al., "Opt: Open pre-trained transformer language models," *arXiv preprint arXiv:2205.01068*, 2022.

[204] J. Hoffmann, S. Borgeaud, A. Mensch, E. Buchatskaya, T. Cai, E. Rutherford, D. d. L. Casas, L. A. Hendricks, J. Welbl, A. Clark et al., "Training compute-optimal large language models," *arXiv preprint arXiv:2203.15556*, 2022.

[205] OpenAI, "Gpt-4 technical report," 2023.

[206] Google, "Palm 2 technical report," 2023. [Online]. Available: https://ai.google/static/documents/palm2techreport.pdf

[207] N. Shazeer, "Fast transformer decoding: One write-head is all you need," *arXiv preprint arXiv:1911.02150*, 2019.

[208] S. Kim, C. Hooper, A. Gholami, Z. Dong, X. Li, S. Shen, M. W. Mahoney, and K. Keutzer, "Squeezellm: Dense-and-sparse quantization," *arXiv preprint arXiv:2306.07629*, 2023.

[209] Y. Leviathan, M. Kalman, and Y. Matias, "Fast inference from transformers via speculative decoding," Nov. 2022.

[210] C. Chen, S. Borgeaud, G. Irving, J.-B. Lespiau, L. Sifre, and J. Jumper, "Accelerating large language model decoding with speculative sampling," Feb. 2023.

[211] H. Xia, T. Ge, S.-Q. Chen, F. Wei, and Z. Sui, "Speculative decoding: Lossless speedup of autoregressive translation," 2023. [Online]. Available: https://openreview.net/forum?id=H-VlwsYvVi

[212] X. Miao, G. Oliaro, Z. Zhang, X. Cheng, Z. Wang, R. Y. Y. Wong, Z. Chen, D. Arfeen, R. Abhyankar, and Z. Jia, "Specinfer: Accelerating generative llm serving with speculative inference and token tree verification," *arXiv preprint arXiv:2305.09781*, 2023.

[213] M. Stern, N. M. Shazeer, and J. Uszkoreit, "Blockwise parallel decoding for deep autoregressive models," *Neural Information Processing Systems*, 2018.

[214] T. Dettmers, A. Pagnoni, A. Holtzman, and L. Zettlemoyer, "Qlora: Efficient finetuning of quantized llms," *arXiv preprint arXiv:2305.14314*, 2023.

[215] L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. L. Wainwright, P. Mishkin, C. Zhang, S. Agarwal, K. Slama, A. Ray et al., "Training language models to follow instructions with human feedback," *arXiv preprint arXiv:2203.02155*, 2022.

[216] W.-L. Chiang, Z. Li, Z. Lin, Y. Sheng, Z. Wu, H. Zhang, L. Zheng, S. Zhuang, Y. Zhuang, J. E. Gonzalez, I. Stoica, and E. P. Xing, "Vicuna: An open-source chatbot impressing gpt-4 with 90%* chatgpt quality," March 2023. [Online]. Available: https://lmsys.org/blog/2023-03-30-vicuna/

[217] J. Ainslie, J. Lee-Thorp, M. de Jong, Y. Zemlyanskiy, F. Lebrón, and S. Sanghai, "Gqa: Training generalized multi-query transformer models from multi-head checkpoints," *arXiv preprint arXiv:2305.13245*, 2023.

[218] R. Pope, S. Douglas, A. Chowdhery, J. Devlin, J. Bradbury, A. Levskaya, J. Heek, K. Xiao, S. Agrawal, and J. Dean, "Efficiently scaling transformer inference," Nov. 2022.

[219] Z. Zhang, Y. Sheng, T. Zhou, T. Chen, L. Zheng, R. Cai, Z. Song, Y. Tian, C. Ré, C. Barrett et al., "H _2 o: Heavy-hitter oracle for efficient generative inference of large language models," *arXiv preprint arXiv:2306.14048*, 2023.

[220] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. E. Gonzalez, H. Zhang, and I. Stoica, "Efficient memory management for large language model serving with pagedattention," in *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*, 2023.

[221] G. Xiao, J. Lin, M. Seznec, H. Wu, J. Demouth, and S. Han, "Smoothquant: Accurate and efficient post-training quantization for large language models," in *International Conference on Machine Learning*. PMLR, 2023, pp. 38 087–38 099.

[222] T. Dettmers, M. Lewis, Y. Belkada, and L. Zettlemoyer, "Llm. int8 (): 8-bit matrix multiplication for transformers at scale," *arXiv preprint arXiv:2208.07339*, 2022.

[223] E. Frantar, S. Ashkboos, T. Hoefler, and D. Alistarh, "Gptq: Accurate post-training quantization for generative pre-trained transformers," *arXiv preprint arXiv:2210.17323*, 2022.

[224] J. Lin, J. Tang, H. Tang, S. Yang, X. Dang, and S. Han, "Awq: Activation-aware weight quantization for llm compression and acceleration," *arXiv preprint arXiv:2306.00978*, 2023.

[225] Y. Xiao, L. Wu, J. Guo, J. Li, M. Zhang, T. Qin, and T.-y. Liu, "A survey on non-autoregressive generation for neural machine translation and beyond," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2023.

[226] B. Spector and C. Re, "Accelerating llm inference with staged speculative decoding," *arXiv preprint arXiv:2308.04623*, 2023.

[227] K. Pillutla, S. Swayamdipta, R. Zellers, J. Thickstun, S. Welleck, Y. Choi, and Z. Harchaoui, "MAUVE: Measuring the gap between neural text and human text using divergence frontiers," in *Advances in Neural Information Processing Systems*, A. Beygelzimer, Y. Dauphin, P. Liang, and J. W. Vaughan, Eds., 2021. [Online]. Available: https://openreview.net/forum?id=Tqx7nJp7PR

[228] A. Holtzman, J. Buys, L. Du, M. Forbes, and Y. Choi, "The curious case of neural text degeneration," in *International Conference on Learning Representations*, 2020. [Online]. Available: https://openreview.net/forum?id=rygGQyrFvH

[229] A. Fan, M. Lewis, and Y. Dauphin, "Hierarchical neural story generation," in *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, 2018.

[230] S. Basu, G. S. Ramachandran, N. S. Keskar, and L. R. Varshney, "{MIROSTAT}: A {neural} {text} {decoding} {algorithm} {that} {directly} {controls} {perplexity}," in *International Conference on Learning Representations*, 2021. [Online]. Available: https://openreview.net/forum?id=W1G1JZEIy5_

[231] C. Meister, G. Wiher, T. Pimentel, and R. Cotterell, "On the probability-quality paradox in language generation," Mar. 2022.

[232] J. Hewitt, C. D. Manning, and P. Liang, "Truncation sampling as language model desmoothing," Oct. 2022.

[233] C. Meister, T. Pimentel, G. Wiher, and R. Cotterell, "Locally typical sampling," *Transactions of the Association for Computational Linguistics*, vol. 11, pp. 102–121, 2023.

[234] S. Elfwing, E. Uchibe, and K. Doya, "Sigmoid-weighted linear units for neural network function approximation in reinforcement learning," *Neural Networks*, 2017.

[235] C. Ying, T. Cai, S. Luo, S. Zheng, G. Ke, D. He, Y. Shen, and T.-Y. Liu, "Do transformers really perform badly for graph representation?" *Advances in Neural Information Processing Systems*, vol. 34, pp. 28 877–28 888, 2021.

[236] A. Kumar, A. Raghunathan, R. Jones, T. Ma, and P. Liang, "Fine-tuning can distort pretrained features and underperform out-of-distribution," *International Conference on Learning Representations*, 2022.

[237] Joao Gante, "Assisted generation: a new direction toward low-latency text generation," 2023. [Online]. Available: https://huggingface.co/blog/assisted-generation

[238] ShareGPT, "ShareGPT," https://huggingface.co/datasets/Aeala/ShareGPT_Vicuna_unfiltered, 2023.

[239] L. Tunstall, E. Beeching, N. Lambert, N. Rajani, K. Rasul, Y. Belkada, S. Huang, L. von Werra, C. Fourrier, N. Habib, N. Sarrazin, O. Sanseviero, A. M. Rush, and T. Wolf, "Zephyr: Direct distillation of lm alignment," 2023.

[240] Y. Kim and A. M. Rush, "Sequence-level knowledge distillation," *EMNLP*, 2016.

[241] E. J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, and W. Chen, "Lora: Low-rank adaptation of large language models," *ICLR*, 2021.

[242] Y. Dubois, X. Li, R. Taori, T. Zhang, I. Gulrajani, J. Ba, C. Guestrin, P. Liang, and T. B. Hashimoto, "Alpacafarm: A simulation framework for methods that learn from human feedback," 2023.

[243] L. Zheng, W.-L. Chiang, Y. Sheng, S. Zhuang, Z. Wu, Y. Zhuang, Z. Lin, Z. Li, D. Li, E. P. Xing, H. Zhang, J. E. Gonzalez, and I. Stoica, "Judging llm-as-a-judge with mt-bench and chatbot arena," 2023.

[244] Axolotl, "Axolotl," https://github.com/OpenAccess-AI-Collective/axolotl, 2023.

[245] T. Dettmers, M. Lewis, S. Shleifer, and L. Zettlemoyer, "8-bit optimizers via block-wise quantization," *International Conference on Learning Representations*, 2021.

[246] N. Ding, Y. Chen, B. Xu, Y. Qin, Z. Zheng, S. Hu, Z. Liu, M. Sun, and B. Zhou, "Enhancing chat language models by scaling high-quality instructional conversations," 2023.

[247] P. Zhang, G. Zeng, T. Wang, and W. Lu, "Tinyllama: An open-source small language model," 2024.

[248] J. Pan, "Tiny vicuna 1b," https://huggingface.co/Jiayi-Pan/Tiny-Vicuna-1B, 2023.

[249] L. Chen, "Dissecting batching effects in gpt inference," https://le.qun.ch/en/blog/2023/05/13/transformer-batching/, 2023, blog.

[250] NVIDIA, "Nvidia a100 tensor core gpu."