UNIVERSITY OF CALIFORNIA SAN DIEGO

Machine Learning Algorithm and System Co-design for Hardware Efficiency

A dissertation submitted in partial satisfaction of the
requirements for the degree Doctor of Philosophy

in

Computer Science

by

Cheng Fu

Committee in charge:

Professor Jishen Zhao, Chair
Professor Ryan Kastner
Professor Farinaz Koushanfar
Professor Jingbo Shang
Professor Hao Su

2023

The Dissertation of Cheng Fu is approved, and it is acceptable in quality and form for publication on microfilm and electronically.

University of California San Diego

2023

DEDICATION

To my friend in heaven, Dr. An Wu.

EPIGRAPH

The struggle itself towards the heights is enough to fill a man's heart.
One must imagine Sisyphus happy.

Albert Camus, The Myth of Sisyphus

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I am grateful for the countless assistance I have received from numerous individuals throughout my Ph.D. studies. I apologize in advance if I inadvertently overlooked any names.

First and foremost, I am indebted to my advisor, Prof. Jishen Zhao, who provided me with the opportunity to pursue my passion at this prestigious institution. Jishen has been incredibly supportive of my career. She has helped me find three internships in both Facebook and Google. She has always trusted the pursuit of my own ideas and has never pressured me into anything that I am not interested in. The past five years of working with her has made me a better researcher.

Also, I would like to express my appreciation to my committee members, Prof. Hao Su, Prof. Jingbo Shang, Prof. Farinaz Koushanfar, and Prof. Ryan Kastner, for their insightful comments and invaluable feedback on improving my dissertation.

The helps from my collaborators have been invaluable in my PhD journey. I extend my sincere thanks to Dr. Huili Chen, Hanxian Huang, Dr. Xinyun Chen, Dr. Shilin Zhu, Prof. Hao Su, Prof. Farinaz Koushanfar, Dr. Yanqi Zhou and Dr. An Wu. I am also grateful to Dr. Yajing Chen, Prof. Hun-seok Kim, and Dr. Ching-En Lee for opening the doors of research when I was a master student at the University of Michigan, Ann Arbor.

I owe immense thanks to my mentors in the industry: Andrew Li and Dr. Sheng Li from Google, Dr. Yuandong Tian and Prof. Hugh Leather from Facebook. Their unwavering passion for research has had a profound influence on me.

To my friends in San Diego, I express my heartfelt appreciation for making me feel at home. I will remember the skiing trip with Dr. Xindong Tang and Jiangchuanhai Wang, as well as the Mario party nights with Xiaoyang Zeng and Zhaoyi Huang. I will also miss chatting with my STABLE lab mates: Zhongming Yu, Hanxian Huang, Zixuan Wang, Haolan Liu, Yun Joon Soh, and Theodoros Michailidis. During my job search, I received invaluable help and emotional support from my friends Yudian Li, Darlene Guo, Tianyi Shan, Zijing Gu, Byung Hoon Ahn, Chenxi Du, and Erik Meade.

I am thankful for all the podcasts, movies, and books that have enlightened my path on

this journey.

My deepest gratitude goes to my parents, Huifang Li and Ruihua Fu, for their unwavering support.

Chapter 2, in part, contains a re-organized reprint of the material as it appears in International Conference on Parallel Architectures and Compilation Techniques (PACT) 2022. Cheng Fu; Hanxian Huang; Bram Wasti; Chris Cummins; Riyadh Baghdadi; Kim Hazelwood; Yuandong Tian; Jishen Zhao; Hugh Leather. The dissertation author was the primary investigator and author of this paper.

Chapter 3, in part, contains a re-organized reprint of the material as it appears in International Conference on Machine Learning (ICML) 2021. Fu, Cheng; Hanxian Huang; Xinyun Chen; Yuandong Tian; Jishen Zhao. The dissertation author was the primary investigator and author of this paper.

Chapter 4, in part, contains a re-organized reprint of the material as it appears in IEEE Micro 2020. Cheng Fu; Huili Chen; Zhenheng Yang; Farinaz Koushanfar; Yuandong Tian; Jishen Zhao. The dissertation author was the primary investigator and author of this paper.

Chapter 5, in part, contains a re-organized reprint of the material as it will appear in International Conference on Computer Vision (ICCV) 2023. Fu, Cheng; Hanxian Huang; Zixuan Jiang; Yun Ni; Lifeng Nai; Gang Wu; Liqun Cheng; Yanqi Zhou; Sheng Li; Andrew Li; Jishen Zhao. The dissertation author was the primary investigator and author of this paper.

VITA

2012–2016 Bachelor of Science, Beijing Institute of Technology

2016–2018 Master of Science, University of Michigan, Ann Arbor

2018–2023 Doctor of Philosophy, University of California San Diego

PUBLICATIONS

C. Fu, H. Huang, Z. Jiang, Y. Ni, L. Nai, G. Wu, L. Cheng, Y. Zhou, S. Li, A. Li, J. Zhao, "TripLe: Revisiting Pretrained Model Reuse and Progressive Learning for Efficient Vision Transformer Scaling and Searching", International Conference on Computer Vision (**ICCV**), 2023

C. Fu, H. Huang, B. Wasti, C. Cummins, R. Baghdadi, K. Hazelwood, Y. Tian, J. Zhao, H. Leather, "Q-gym: An Equality Saturation Framework for DNN Inference Exploiting Weight Repetition", International Conference on Parallel Architectures and Compilation Techniques (**PACT**), 2022.

H. Chen, **C. Fu**, J. Zhao, F. Koushanfar. "GALU: A Genetic Algorithm Framework for Logic Unlocking", Digital Threats: Research and Practice.

H. Chen, **C. Fu**, J. Zhao, F. Koushanfar. "ProFlip: Targeted Trojan Attack with Progressive Bit Flips", International Conference on Computer Vision **(ICCV)**, 2021

C. Fu, H. Huang, X. Chen, Y. Tian, J. Zhao. "Learn-to-Share: A Hardware-friendly Transfer Learning Framework Exploiting Computation and Parameter Sharing", International Conference on Machine Learning **(ICML)**, 2021 **(long oral)**

C. Fu, H. Chen, Z. Yang, H. Liu, F. Koushanfar, Y. Tian, J. Zhao. "Enhancing Model Parallelism in Neural Architecture Search for Multi-device System", IEEE Micro, 2020

C. Fu, H. Chen, H. Liu, X. Chen, Y. Tian, F. Koushanfar, J. Zhao. "Coda: An End-to-End Neural Program Decompiler", Neural Information Processing System **(NeurIPS)**, 2019

H. Chen, **C. Fu**, J. Zhao, and F. Koushanfar, "GenUnlock: An Automated Genetic Algorithm Framework for Unlocking Logic Encryption", International Conference on Computer Aided Design **(ICCAD)**, 2019 **(Best Paper Nominee)**

H. Chen, **C. Fu**, J. Zhao, and F. Koushanfar, "DeepInspect: A Black-box Trojan Detection and Mitigation Framework for Deep Neural Networks", International Joint Conference on Artificial Intelligence **(IJCAI)**, 2019.

H. Chen, **C. Fu**, B. D. Rouhani, J. Zhao, and F. Koushanfar, "DeepAttest: An End-to-End Attestation Framework for Deep Neural Networks", International Symposium on Computer Architecture **(ISCA)**, 2019.

C. Fu, S. Zhu, H. Su, C. Lee, and J. Zhao. "Towards Fast and Energy-Efficient Binarized Neural Network Inference on FPGA.", International Symposium on Field-Programmable Gate Arrays **(FPGA)**, 2019

H. Chen, B. D. Rouhani, **C. Fu**, J. Zhao, and F. Koushanfar, "DeepMarks: A Secure Fingerprinting Framework for Digital Rights Management of Deep Learning Models", International Conference on Multimedia Retrieval **(ICMR)**, 2019

C. Fu, A. Di Fulvio, S.D. Clarke, D. Wentzloff, S.A. Pozzi, H.S. Kim, "Artificial neural network algorithms for pulse shape discrimination and recovery of piled-up pulses in organic scintillators," in Annals of Nuclear Energy, 2018

Y. Chen, S. Lu, **C. Fu**, D. Blaauw, R. Dreslinski Jr, T. Mudge, and H. Kim, "A Programmable Galois Field Processor for the Internet of Things," International Symposium on Computer Architecture **(ISCA)**, 2017

Y. Lin, Z. Jiang, **C. Fu**, H. K.H. So, and H. Yang. FPGA High-level Synthesis versus Overlay: Comparisons on Computation Kernels. Int'l Symp. on Highly-Efficient Accelerator and Reconfigurable Technologies (HEART), 2017

ABSTRACT OF THE DISSERTATION

Machine Learning Algorithm and System Co-design for Hardware Efficiency

by

Cheng Fu

Doctor of Philosophy in Computer Science

University of California San Diego, 2023

Professor Jishen Zhao, Chair

Deep Neural Networks (DNNs) are increasingly adopted in various fields due to their unprecedented performance. Yet, the computation overhead of DNN evaluation and training continues to grow exponentially. Enormous system-level advancements have recently been witnessed to improve the efficiency of DNN computation; many efficient DNN algorithms are proposed to reduce the cost of DNN computation. However, this is far from optimal. Most current DNN computation systems do not fully exploit efficient ML algorithms, while ML algorithms fail to consider novel systems for deployment.

This thesis focuses on designing efficient DNN algorithms and DNN computation systems to enable fast DNN training and evaluation. Instead of solely focusing on creating either new

DNN algorithms or systems, we propose a co-design approach by exploring DNN models that are system-aware and DNN computation systems that are algorithm-aware. By leveraging such a co-design approach, we effectively advance the Pareto-frontier between task accuracy and efficiency of DNN execution in various application domains.

We present a set of works that explore the co-design method. Firstly, we present an algorithm-aware DNN compiler for quantized DNN. By leveraging the weight repetition feature of this efficient DNN algorithm, we can greatly reduce the computation overhead of DNN inference on both CPU and GPU. Secondly, we discuss a hardware-aware DNN algorithm with enhanced model parallelism. We observe that previous works design efficient DNNs for single-device platforms. When customizing the DNN design for a multi-device system, we can reduce the DNN inference latency by a large margin while previous models can hardly be parallelized across multiple devices. Thirdly, we present a hardware-friendly transfer learning framework for natural language processing tasks. The existing transfer learning frameworks have a lot of computation redundancy when deploying on the existing systems. By reusing the computation of different transfer learning models, we can greatly reduce the computation overhead as well. Lastly, we introduce a novel training method to reduce the computation cost of DNN training and DNN design process. The key idea is to initialize the large models using small pretrained weights. The implicit knowledge in the pretrained models facilitates faster convergence of the large models. Besides, changing only the initialization phase of training means no extra computation overhead will be introduced to the existing training systems. Also, this new training method can be applied to accelerate the design process of system-aware DNN models.

As Moore's Law is slowing down, the computational capacity of current DNN systems is plateauing. This thesis sheds light on how to overcome this limitation by designing domain-specific DNN algorithms and computation systems.

Chapter 1

Introduction

1.1 Challenge

Deep Neural Networks (DNNs) have completely changed our way of living, ushering in remarkable advancements in various fields, such as computer vision [57, 92, 39, 121], natural language processing [38, 16, 155], medical diagnosis [169]. Many widely-used commercialized AI tools powered by DNNs are developed, such as conversational agents ChatGPT[1], code assistant tool Copilot[2], and text-to-image generation tool Midjourney[3]. These cutting-edge applications are gradually becoming tools that people use every day for work. The development of AI is the result of not only the rapid development of DNN algorithms but also significant progress in DNN computation systems.

However, along with the superior performance, DNNs also bring certain challenges, particularly regarding computation overhead. Over the past decade, the computation required for training and deploying DNNs has experienced exponential growth, as illustrated in Figure 1.1[4]. The computation overhead of DNNs has increased more than $300,000\times$ since 2012 and this trend is still continuing. The increasing computational demands can lead to scalability issues, preventing further performance improvements in ML technology. Also, the computation overhead

[1] https://chat.openai.com/
[2] https://github.com/features/copilot
[3] https://www.midjourney.com/
[4] https://openai.com/research/ai-and-compute

of DNNs poses considerable time and energy consumption, making it even more harder and expensive to train and deploy sophisticated DNN models.

The increasing complexity of DNN architectures and the ever-expanding size of datasets have contributed to this trend. The computation overhead comes from both the DNNs training and inference. First, the training of large DNNs demands immense computational power, often necessitating specialized hardware and compilers, such as Graphics Processing Units (GPUs) with NVIDIA CUDA compiler [110] and Tensor Processing Units (TPUs) [74] with XLA compiler [130]. Second, the inference phase, where the trained model makes predictions on new data during online deployment, also demands significant computational resources. The computation of a single query shows a smaller overhead compared to model training. However, for commercialized AI tools with million users, an extensive amount of queries from users happen every hour. In this scenario, the inference latency is also critical.

1.2 Solution

Machine Learning (ML) researchers have devised efficient DNN algorithms for training [95, 79, 21] and inference to reduce the computation overhead of DNN. For DNN inference, two common approaches are leveraging weight sparsity [56, 66] and quantization [158, 184, 69]. However, these approaches typically incur DNN performance loss. Depending on the degree of DNN performance drop, the computation latency of the DNN varies, resulting in a Pareto-curve between the execution latency and task performance (Figure 1.3).

Also, system researchers have developed general-purpose ML compilers [123, 24, 130] and hardware [74, 26, 136] to reduce DNN execution latency in both training and inference.

When the efficiency gains from the novel DNN algorithm and new DNN computation systems reach a plateau, we investigate a new method to further reduce the overhead of DNN computation and to advance the frontier of the Pareto-curve as shown in Figure 1.3. Specifically, we leverage the co-design approach summarized in Figure 1.2 (b). When the hardware for DNN

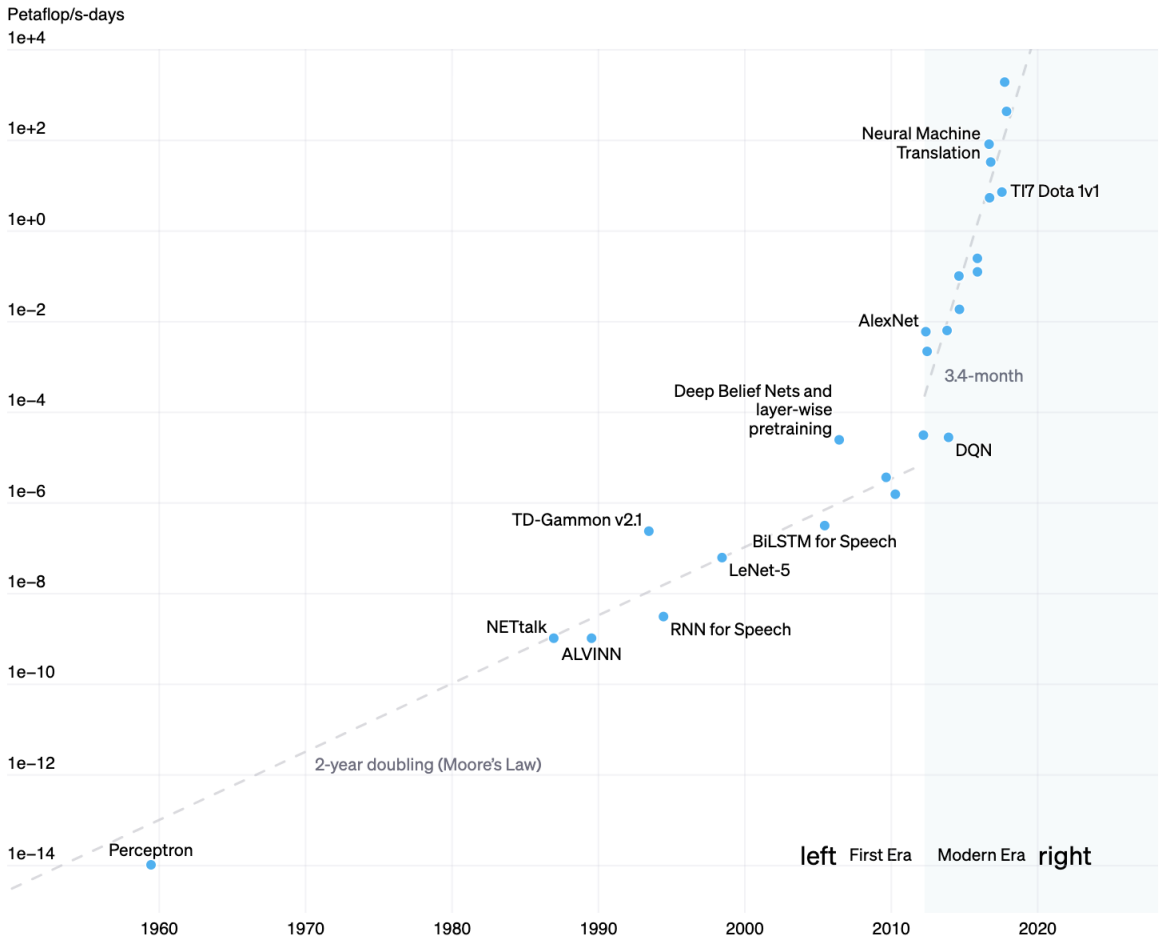


Figure 1.1. Two eras of computing usage in training AI systems.

deployment is known before the DNN design process, we can exploit this information to tailor the DNN design for a specific computation environment. Similarly, we can customize the design of a system (i.e., hardware or compiler) to suit a specific efficient ML algorithm.

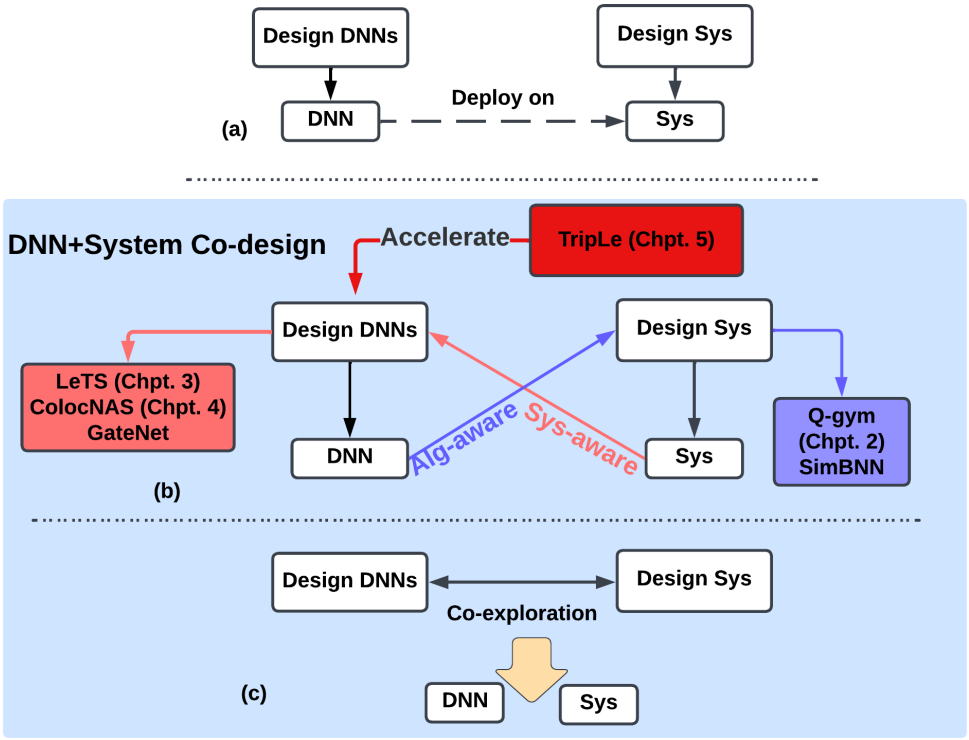


Figure 1.2. (a) Traditional design process of DNN computation system and DNN. (b) The design process of algorithm-aware system design and system-aware DNN model design. (c) Co-exploration of DNN design and system design.

Algorithm-aware System Design. Instead of creating general-purpose hardware or DNN compilers, we can customize the system design for a specific efficient ML algorithm, such as designing domain-specific accelerators that leverage DNN sparsity or quantization [44]. EIE [55] exploits DNN sparsity and Huffman coding to accelerate sparse model inference. BiTFusion [137] leverages low bitwidth multipliers for accelerating quantized neural networks (QNNs) [176, 184]. In this dissertation, we present a DNN compiler that generates model-specific computation dataflow for different QNNs (Chapter I).

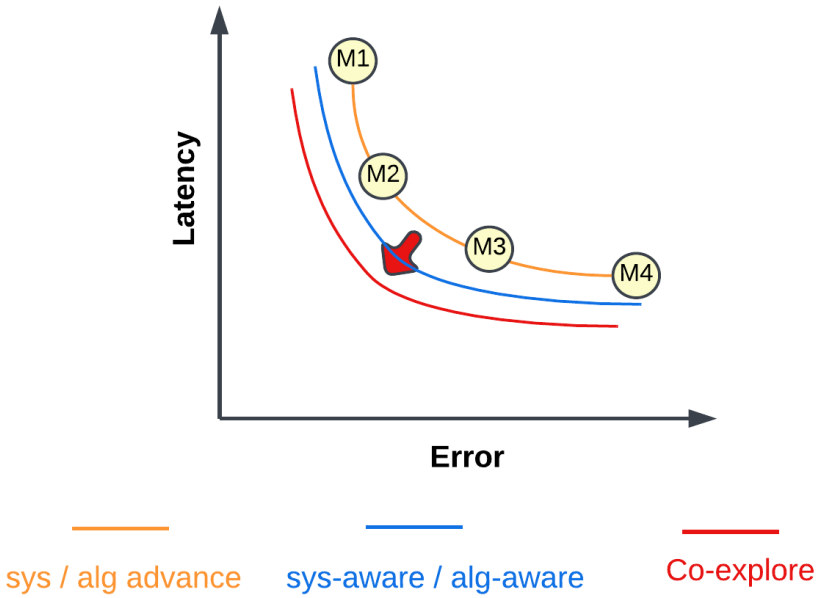


Figure 1.3. 'M1'-'M4' denote different DNNs. By leveraging the co-design approach (i.e., system-aware algorithm design, algorithm-aware system design, and co-exploration) we can push the frontier of Pareto-curve between task performance and DNN runtime latency.

System-aware DNN Algorithm Design. When designing DNN models, ML researchers typically apply general hardware cost metrics to measure the model computation overhead, such as *parameter size* and *FLOPs*[5]. In reality, those metrics are not good indicators of hardware efficiency due to oversimplification. For this reason, we propose to design the DNN models (Chapters 3 and 4) that consider the computation cost of system design during deployment.

As discussed in Sec. 1.1, the training phase of DNNs is also computationally intensive. It poses a large computation overhead during the DNN model design process. In Chapter 5, we propose TripLe, a DNN training method that can accelerate the design process of system-aware DNNs. This method also considers the system design for training by introducing no extra computation overhead while significantly reducing the training time of DNNs.

Co-exploring DNNs and system designs. Our eventual goal is to design DNNs together with the system. Previous work [183] has leveraged reinforcement learning [134] (RL) to sample

[5]FLOPs in this dissertation refers to the total number of adds and multiplies for computing a DNN.

models and hardware design parameters together. However, the DNN design space and the system design space are small in these works due to the limitation of the sample efficiency of RL algorithms. We left the investigation of co-exploring DNN and DNN computation systems as future works to further push the Pareto curve.

Table 1.1. Summary of each work in ML algorithm and system design perspective.

	Domain	ML algorithm	System
Q-gym	DNN inference for image classification	DNN Quantization	New DNN compiler that exploits weight repetition features of quantized DNNs
LeTS	Accelerate transformer inference when multiple subtasks co-exist	New searching algorithm to maximize the computation reuse	Design transfer learning models and systems that can reuse computations
ColocNAS	DNN inference for image classification	New searching algorithms to find parallelizable DNNs	Latency prediction for DNNs on multi-device computation systems
TripLe	Transformer training and searching	Exploiting existing pretrained models	-

1.3 Thesis Contributions

This section provides an overview of this dissertation. The dissertation includes four works that increase hardware efficiency for either DNN inference or training under different computation environments. For each work, we consider both efficient ML algorithms and the features of the deploying systems to push the Pareto-frontier between hardware latency and task performance. The summary of each work in terms of system and ML perspectives is given in Table 1.1.

Chapter 2: An Algorithm-aware Equality Saturation Framework for DNN Inference Exploiting Weight Repetition. Prior works employed weight repetition in quantized neural networks to save the computation of convolutions by memorizing or arithmetic factorization. However, such methods fail to fully exploit the exponential search space by factorizing and reusing computation. We propose Q-gym, a DNN framework consisting of two components. First, we propose a compiler, which leverages equality saturation to generate computation expressions for convolutional layers with a significant reduction in the number of operations. Second, we integrate the computation expressions with various parallelization methods and homomorphic encryption DNN evaluation flow to accelerate a set of downstream tasks. Extensive experiments show that Q-gym achieves 19.1% / 68.9% more operation reductions compared to SumMerge and original DNNs. Also, computation expressions from Q-gym contribute to 2.56× / 1.78× inference speedup on CPU / GPU compared to OneDNN and PyTorch GPU on average. Furthermore, Q-gym reduces the Homomorphic operations by 2.47× / 1.30× relative to CryptoNet and FastCryptoNet with only 0.06% accuracy loss due to quantization.

Chapter 3: A Hardware-friendly Transfer Learning Framework Exploiting Computation and Parameter Sharing. Task-specific fine-tuning on pre-trained transformers has achieved performance breakthroughs in multiple NLP tasks. Yet, as both computation and parameter size grows linearly with the number of sub-tasks, it is increasingly difficult to adopt such methods to the real world due to unrealistic memory and computation overhead on computing devices.

Previous works on fine-tuning focus on reducing the growing parameter size to save storage cost by parameter sharing. However, compared to storage, the constraint of computation is a more critical issue with fine-tuning models in modern computing environments. We propose *LeTS*, a framework that leverages both computation and parameter sharing across multiple tasks. Compared to traditional fine-tuning, LeTS proposes a novel neural architecture that contains a fixed pre-trained transformer model, plus learnable additive components for sub-tasks. The learnable components reuse the intermediate activations in the fixed pre-trained model, decoupling computation dependency. Differentiable neural architecture search is used to determine a task-specific computation sharing scheme, and a novel early stage pruning is applied to additive components for sparsity to achieve parameter sharing. Extensive experiments show that with 1.4% of extra parameters per task, LeTS reduces the computation by 49.5% on GLUE benchmarks with only 0.2% accuracy loss compared to full fine-tuning.

Chapter 4: Designing DNNs with Model Parallelism for Multi-device System. Neural architecture search (NAS) finds favorable network topologies for better task performance. Existing hardware-aware NAS techniques only target to reduce inference latency on single CPU/GPU systems and the searched model can hardly be parallelized. To address this issue, we propose *ColocNAS*, the first **synchronization-aware**, **end-to-end** NAS framework that automates the design of parallelizable neural networks for multi-device systems while maintaining a high task accuracy. ColocNAS defines a **new search space** with elaborated connectivity to reduce device communication and synchronization. ColocNAS consists of three phases: (i) Offline latency profiling that constructs a lookup table of inference latency of various networks for online runtime approximation. (ii) Differentiable latency-aware NAS that simultaneously minimizes inference latency and task error. (iii) Reinforcement learning (RL)-based device placement fine-tuning to further reduce the latency of the deployed model. Extensive evaluation corroborates ColocNAS's effectiveness to reduce inference latency while preserving task accuracy.

Chapter 5: Accelerating DNN Training and Searching by Reusing Pretrained Model and Progressive Learning. One promising way to accelerate transformer training is to reuse small

pretrained models to initialize the transformer, as their existing representation power facilitates faster model convergence. Previous works designed expansion operators to scale up pretrained models to the target model before training. Yet, model functionality is difficult to preserve when scaling a transformer in all dimensions at once. Moreover, maintaining the pretrained optimizer states for weights is critical for model scaling, whereas the new weights added during expansion lack these states in pretrained models. To address these issues, we propose *TripLe*, which partially scales a model before training, while growing the rest of the new parameters during training by copying both the warmed-up weights with the optimizer states from existing weights. As such, the new parameters introduced during training will obtain their training states. Furthermore, through serializing the model scaling, the functionality of each expansion can be preserved. We evaluate TripLe in both single-trial model scaling and multi-trial neural architecture search (NAS). Due to the fast training convergence of TripLe, the proxy accuracy from TripLe better reveals the model quality compared to from-scratch training in multi-trial NAS. Experiments show that TripLe outperforms from-scratch training and knowledge distillation (KD) in both training time and task performance. TripLe can also be combined with KD to achieve an even higher task accuracy. For NAS, the model obtained from TripLe outperforms DeiT-B in task accuracy with 69% reduction in parameter size and FLOPs.

Chapter 2

An Algorithm-aware Equality Saturation Framework for DNN Inference Exploiting Weight Repetition

2.1 Introduction

Deep Neural Networks (DNNs) are becoming the *de-facto* solutions for various computer vision tasks [36, 92]. However, due to the large computation and storage cost of convolutional layers, DNNs are difficult to integrate with many computation environments, such as mobile devices. To mitigate the computation and storage constraints for efficient DNN deployment, mainstream approaches include Quantizing DNNs [180, 177, 184] and leveraging DNN sparsity [102, 104, 65].

One key characteristic of quantized DNNs (QNNs) is *weight repetition*, i.e., weights in different layers are repetitions of a small number of Q unique ones (e.g., $\{-0.18, 0, 0.18\}$ and $Q = 3$). Leveraging such a regularity, a state-of-the-art approach SumMerge [117] reduces computation by two measures: **i)** Factorization – for example, factorizing a dot-product $ax + by + az + aw$ to be $a(x + z + w) + by$ and **ii)** Computation reuse – for instance, when computing two dot products $r_0 = ax + ay + az + aw$ and $r_1 = bx + ay + bz + bw$, SumMerge first calculates a *partial sum* $e_0 = x + z + w$ and reuses it in the downstream computations ($r_0 = ae_0 + ay$, $r_1 = be_0 + ay$).

While SumMerge indeed reduces the computation, it may not fully exploit weight

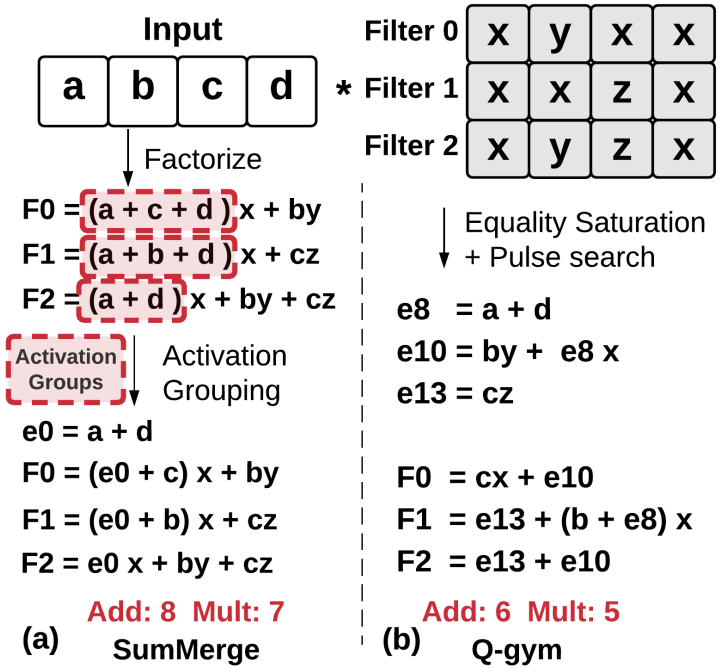


Figure 2.1. Comparison between Q-gym and a state-of-the-art approach SumMerge [117] with an example.**(a)** SumMerge first factorizes each individual dot-product and then performs greedy searches for the partial sum that can be reused by different activation groups. Partial products ($b \cdot y$, $c \cdot z$) are not reused. **(b)** Q-gym leverages both partial sums and partial products (e.g., $e10$ as a partial sum and product, is reused in the follow-up computation of $F0$ and $F2$).

repetition. As shown in Figure 2.1, while SumMerge leverages partial sums, it cannot reuse *partial products* (e.g., *by*, *cz*), leading to sub-optimal reduction. The underlying reason is straightforward: finding the best strategy to reduce the cost of an arithmetic expression is a combinatorial optimization problem, and the simple greedy heuristics leveraged by SumMerge may be insufficient.

Another technique for reducing the cost of convolutions is Winograd [87] which uses fast FFT to reduce computation operations in convolutional layers. Although this can have a significant effect on the number of operations, it does not take advantage of weight reuse and so does not achieve optimal reduction.

We propose *Q-gym*, a framework to accelerate DNN inference by fully exploiting weight repetition. Q-gym comes with two components: a *compiler* that finds smart ways to reduce

addition and multiplication counts in DNN evaluation, and a set of *downstream tasks* that leverage the compiled output to reduce its wall-clock time.

Q-gym's Compiler. To reuse both partial sum and product results, our compiler aggressively explores the search space by leveraging a data structure called *e-graph* [108, 37] that represents a large number of equivalent expressions in a compact manner (Figure 2.2). Thanks to its compact representation, the search space is easier to navigate. To identify the optimal solution in the search space, our compiler performs three steps. (i) Following *equality saturation* [167], we expand the e-graph by repeatedly applying a set of rewrite rules (Table 2.2). (ii) Then, we extract the optimal computation expressions using integer linear programming (ILP) [133]. (iii) With large convolutional layers, e-graph expansion hardly saturates. In order to fully explore the search space with such layers, we employ a *pulsed searching* [106] algorithm, which iteratively applies the first two steps until convergence of the computation cost. In the simple example illustrated in Figure 2.1, our compiler identifies its optimal solution. For large convolution layers, Q-gym can still find more optimized solutions compared to the greedy-based method.

We also discover that the input activations overlap during the 'sliding' computation of weight kernels. That means computations can be reused between the sliding convolutions. With the proposed powerful searching algorithm, our compiler identifies a better solution by leveraging an extended search space compared with SumMerge [117] and Winograd [87].

Q-gym's Downstream Tasks. With the reduced DNN operations generated by our compiler, we can accelerate multiple downstream tasks, such as DNN evaluation on CPU/GPU and DNN evaluation under Homomorphic Encryption (HE) [48].

We propose an acceleration scheme to exploit the parallelism and locality in CPU/GPU to coupe with Q-gym's efficient expressions. In particular, by replacing the inner-dot product of a convolution layer with our expressions, we can exploit the reduction in operations to yield speedup. With different configurations in parallelization, our acceleration scheme can fully utilize the computation resource available. Our acceleration scheme can also effectively integrate with other parallelization schemes, such as loop tiling, vectorization, and multithreading.

We also propose to use Q-gym's efficient expressions for DNN evaluation under HE to preserve data privacy. It is compatible with mainstream HE libraries while significantly reducing the evaluation overhead.

Experiments. Extensive experiments show that our framework effectively accelerates various DNN applications. For QNN where the number of unique weight (Q) $Q \leq 12$, Q-gym reduces the number of FLOPs (#FLOPs) by 68.9% / 19.1% compared to naïve QNN / SumMerge on average. On CPU ($Q \leq 3$), Q-gym achieves a speedup of 1.83× / 2.56× compared to SumMerge / OneDNN [2]. On GPU ($Q \leq 3$), Q-gym achieves a speedup of between 1.64× / 1.78× relative to SumMerge / PyTorch GPU [112]. For DNN evaluation under HE, Q-gym shows 59.5% / 22.9% HE operation reduction compared to CryptoNet [48] and Faster CryptoNet [27] with only -0.06% accuracy reduction due to quantization.

Summary. We summarize our contributions as follows:

- We propose Q-gym, a framework for quantized neural networks that can yield efficient computation dataflow for various downstream applications.
- Q-gym leverages an iterative searching algorithm, e-graph representation, ILP formulation, and elaborated search space to accelerate QNN inference.
- We implement back-end frameworks of Q-gym for multiple sub-domains for QNNs deployment tasks, such as CPU / GPU inference and HE applications. We combine Q-gym with other acceleration schemes and HE protocols.
- We corroborate Q-gym's general applicability and superior performance across various quantized DNN models.

Q-gym is the first framework to fully exploit weight repetition to accelerate privacy-preserving DNN inference. Also, this is the first work that applies the idea of equality saturation to simplify the arithmetic of DNN computation.

2.2 Notations and Background

Notations of CNN. While Q-gym is broadly applicable to any DNNs that can be represented by dot-products, we focus on deep convolutional neural networks [57, 88] (referred to as DNNs in the rest of this section) as examples in this section to make our proposal more concrete.

DNNs are commonly used for image classification [84, 57], object detection [128, 127], and image segmentation [129]. A convolutional layer implements a set of weight kernels to detect features in the input image. A weight kernel is defined by a set of weights, W, and a bias term, B. Each convolutional layer applies K kernels of dimension $R \times S \times C$ on the input with dimension $H \times W \times C$, resulting in output feature maps with dimension $(H-R+1) \times (W-S+1) \times K$. C denotes the number of input channels. H and W denote input height and weight. $R \times S$ denotes kernel shape. Formally, suppose the input activation is IA and weight is L, then the output activation O of this convolution operation is the dot product between input IA and parameters L, added by the bias B, and followed by a non-linear activation function $g(\cdot)$. For a layer with a unit stride, this can be represented as:

$$O(x,y,k) = g(B + \sum_{c=0}^{C-1} \sum_{r=0}^{R-1} \sum_{s=0}^{S-1} L(r,s,c,k) \cdot IA(x+r, y+s, c)) \tag{2.1}$$

For a quantized DNN (QNN), we denote the number of unique weights as Q. For a sparse DNN, we denote the sparsity ratio – the ratio between the number of non-zero weights and the number of parameters – as Sp.

Background on Homomorphic Encryptions. A broad range of applications, such as medical [77, 119], and fraud detection [5, 114], require privacy and confidentiality in client data. Homomorphic Encryption (HE) [47] is an ideal solution for such applications. When using HE for DNN inference, the *Model vendors* provide DNN model M and evaluation function f that computes $\square$ which satisfies $E(x_i) \square M = E(f(x_i, M), k_{pub})$. After the *cloud* completes the model inference, the client can decrypt (D) the result using the private key k_{pri} through Eq.(2.2). In this

Table 2.1. Comparison between methods that reduce QNN computation costs by leveraging weight repetitions. *The temporal reuse scheme is described in Section 2.4.5.

Methods	Algorithms		
	AGR	SumMerge	Q-Gym
Evaluated Hardware	CPU	CPU	CPU+GPU+HE
HE Application	×	×	✓
Temporal Reuse*	×	×	✓
Bypass partial product	×	×	✓
Average FLOPs Reduction	<49.8%	49.8%	**68.9%**
Max CPU/GPU Speedup	-	3.1× / 1.6×	**5.9× / 3.1×**

way, both parties can keep their data private during the DNN evaluation.

$$f(x_i, M) = D(E(f(x_i, M), k_{pub}), k_{pri})). \tag{2.2}$$

The key bottleneck of applying HE on DNN is that homomorphic multiplications and additions are extremely computationally expensive [76, 27]. An evaluation of CryptoNet [48] requires 250 seconds on the MINST dataset. The inference time of DNN under HE is proportional to the number of HE operations [27] (Detailed in Sec. 2.5.2).

In this section, we identify that exploiting weight repetition can greatly reduce the computation overhead for popular HE protocols. The efficient evaluation function f from Q-gym reduces the number of HE operations by up to 2.47× / 1.30× compared to the state-of-the-art methods (FastCryptoNet/CryptoNet).

2.3 Motivation of Q-gym

In this section, we discuss the motivations for proposing Q-gym.

Weight Quantization is Not Fully Explored. Weight quantization is a widely used method to reduce the storage and computation overhead for deploying DNNs (See Section 2.7). Most existing works for QNN acceleration, such as BitFusion [137] and FLIM [144], leverage the shorter bit width of weights to accelerate each multiplication and addition. However, we identify that weight repetition features in QNNs are not fully explored for computation reduction in

existing works. Specifically, we can bypass computations by factorizing the computation or reusing partial results. (Figure 2.1).

Previous Works Employing Weight Repetition is Sub-optimal. Activation group reuse (AGR) [58] is the first method that exploits weight repetition for efficient inference. The dot-product between input and each weight kernel will be factorized into expressions with reduced multiplications (Figure 2.1(a)). The operands of factorized expressions formed *activation groups* and common sub-expressions between groups can be shared to reduce computation. Starting from the first activation group, AGR greedily extracts the maximum overlapping term between the first and the rest of the terms in the activation groups. Then, it recursively searches the second term and so on.

SumMerge [117] also adopts factorization and activation grouping schemes. Different from AGR, SumMerge (1) iterates all activation groups pairs and generates shared 'sub-computation' terms between activation groups in the activation grouping phase and (2) employs a "maxscore" – defined as the number of times each 'sub-computation' term appears across activation groups – to determine the sub-computation terms to be reused across. SumMerge iterates between (1) and (2) until no shared sub-computation terms are left.

However, SumMerge is inefficient in the following aspects. i) SumMerge is still a greedy heuristic. Extracting the sub-computation term with the best 'maxscore' in each iteration may not be the optimal solution. ii) Factorization can guarantee an upper bound on multiplications ($Q \cdot K$) for the convolution layer L. However, the search space of SumMerge is limited, as it only searches for common sub-expressions between activation groups. In other words, SumMerge does not fully exploit the search space of potential computation reuse. For example, it does not reuse the computed result to bypass partial products (e.g., *by*, *cz* in Figure 2.1). Also, with the increase of Q, the performance of SumMerge drops drastically (Section 2.6.1) as the size of each activation group reduces. iii) With the increasing size of weight kernels, the computation of "maxscore" is extremely expensive and incurs a large compilation time. In Figure 2.1, SumMerge and AGR yield the same results. In Table 2.1, we summarize the comparisons between Q-gym

and AGR/SumMerge.

Applying Weight Repetition in Acceleration is Non-trivial. As shown in Eq.(2.1), the computation of a convolution layer involves 6 loops (H, W, R, S, C, K), and how to parallelize the computation has been a long-time problem [123, **?**, 137]. The most common techniques involve 1) using loop tiling [137] to get better data locality. 2) Multi-threading and 3) SIMD vectorization.

These techniques cannot be directly used in our scenario as Q-gym changes the computation dataflow between the input and weight kernels. In this section, we carefully design our parallelization scheme so that we can combine the efficient arithmetic compiled from Q-gym with the common acceleration methods on general-purpose hardware (Detailed in Sec. 2.5.1).

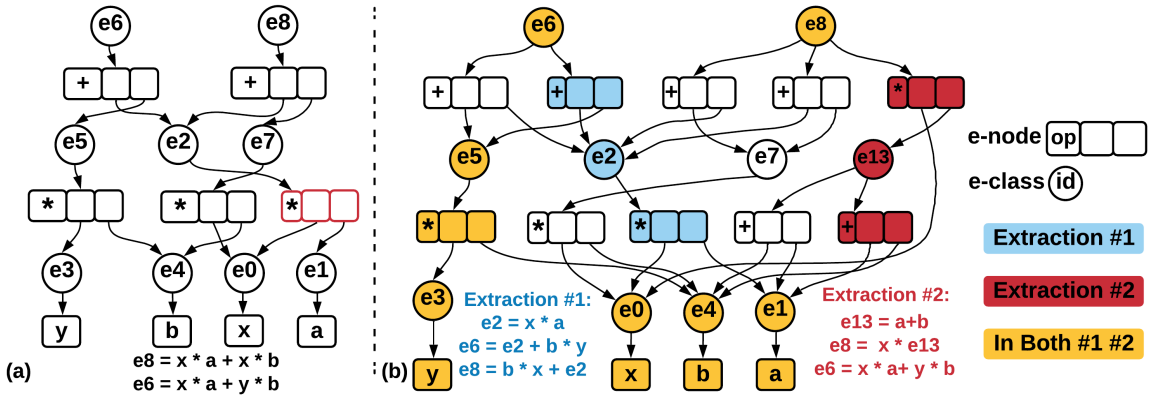


Figure 2.2. An example of (a) e-graph initialization and (b) e-graph after expansion using rules in Table 2.2. With different e-class / e-node selected, we can extract different equivalent computation expressions from an expanded e-graph.

2.4 Q-gym: Compiler Design

2.4.1 Overview

As discussed in Section 2.3, the previous method of SumMerge [117] with a greedy heuristic cannot fully explore the search space and fails to maximize computation reuse. We propose Q-gym that leverages equality saturation (Section 2.4.2) to resolve the limitations of SumMerge. Q-gym's compiler incorporates a two-phase design: *exploration* and *extraction*. During the exploration stage (Section 2.4.3), Q-gym applies a set of rewrite rules on the computation expressions represented in an efficient data structure that can compactly represent a large number of equivalent expressions. For the extraction stage (Section 2.4.4), Q-gym formulates the selection of expressions as an ILP problem and finds more low-cost solutions compared to the greedy method. To handle an even larger search space, Q-gym also uses a *pulsed searching* algorithm [106] that iterates the *exploration* and *extraction* until convergence (Section 2.4.5). We also observe that inputs are overlapped during the sliding window computation of output. As such, we propose a *temporal reuse* search space that can further reduce the operations (Section 2.4.5).

2.4.2 Equality Saturation

Equality saturation [142, 162] is a method to resolve the exponential time and space requirement of traditional graph rewriting. Leveraging the efficient data structure *e-graph* underlying the equality saturation, the rewriting can be applied to the e-graph simultaneously without interfering with each other. Also, the e-graph is a compact representation of equivalent representations, which resolves the memory space limitation in traditional rewriting. In this section, we propose to use the idea of equality saturation to reduce QNN operations. Many other applications leveraging equality saturation are discussed in Section 2.7.

In this subsection, we introduce the e-graph and the rewriting rules applied in Q-gym.

E-graphs. An e-graph is a data structure that compactly represents equivalent expressions. An e-graph contains a set of *e-classes* and a set of *e-nodes*. Each e-class represents a set of equivalent

Table 2.2. Rewrite Rules used in Q-gym for DNN arithmetic simplification. e_i denotes an e-class.

Description	Source	Target
Mult Commutative	$e_i \times e_j$	$e_j \times e_i$
Add Commutative	$e_i + e_j$	$e_j + e_i$
Add Associative I	$(e_i + e_j) + e_k$	$e_i + (e_j + e_k)$
Add Associative II	$e_i + (e_j + e_k)$	$(e_i + e_j) + e_k$
Mult Distributive	$(e_i + e_j) \times e_k$	$e_i \times e_k + e_j \times e_k$
Mult Factorise	$e_i \times e_k + e_j \times e_k$	$(e_i + e_j) \times e_k$

terms that can be computed from any of its e-node children. Each root e-class represents the final computed term from the input and a weight kernel.

Figure 2.2(a) shows an e-graph that represents the computation of an input ($[a,b]$) and two simple weight kernels ($[x,x]$ and $[x,y]$). The expressions that compute these computations are given at the bottom of the graph, defining e8 and e6 respectively. After applying rule rewriting (Table 2.2) to *saturation* (defined in Section 2.4.3), the e-graph will be expanded to Figure 2.2(b).

In Q-gym, an e-node is one operator (i.e., '*' and '+') associated with two operands. The input value of each e-node's operand is a child e-class. Formally,

(i) *An e-graph represents a term if any of its e-classes do.*

(ii) *An e-class represents a term if any of its equivalent child e-nodes do. All terms represented by an e-class are equivalent.*

(iii) *An e-node $f(c_0,c_1)$ represents a term $f(e_0,e_1)$ if each e-class e_i represents c_i.*

In Figure 2.2(b), 'e13' is an e-class that represents both $b+a$ and $a+b$. In Figure 2.2(a), the e-node marked in red represents the term $x*a$ as e-classes e_0 and e_1 represent x and a.

Rewrite Rules. In equality saturation, we expand the egraph using a set of rewrite rules given in Table 2.2. The set of rewrite rules states the equivalence between each pair of computation arithmetic expressions. During e-graph exploration, rewrite rules will be applied to the e-graph. Specifically, we search for a pattern (source pattern) in the input e-graph that is equivalent to another subgraph pattern (target pattern). Q-gym begins rewriting the e-graph after all the applicable rules are searched.

Note that some of the rewrite rules do not create any new e-class to the e-graph (e.g. the commutative rules). However, they may allow other rules to be applied in the computation (e.g. factorization). Also, these rules increase the connections between e-nodes and e-classes, which enables the extraction phase to find better expressions for computation reuse.

2.4.3 Exploration Phase

As discussed in Section 2.4.1, in the first stage of equality saturation, we expand the e-graph using a set of rewrite rules. For example, the exploration phase expands the e-graph from Figure 2.2(a) to Figure 2.2(b).

An e-graph is initialized from the input expressions (Figure 2.2(a)). Then, we search the source pattern of each rewrite rule in the e-graph. If a match is found, it returns the e-class and corresponding substitution (target pattern) to the 'match list' U.

Once all the (e-class, substitution) pairs are found, we begin to modify the e-graph in the 'match list' simultaneously. Each pair of modifications (i.e., e-class and substitution pair) may create a new e-class/e-node in the graph unless the e-class/e-node already exists.

An e-graph is called *saturated* if no more e-class/e-node can be created. The exploration phase will stop when one of the following three conditions is met: 1) The e-graph is saturated. 2) The maximum exploration iterations limit $max\_explore\_iter$ is reached. 3) The maximum e-node limit max_{enode} is reached. Note that the e-graph for Q-gym is finite though possibly very large. Long-running explorations are prevented by $max\_explore\_iter$ and max_{enode}. The pseudo-code of the exploration phase is shown in Algorithm 1.

2.4.4 Extraction Phase

After the exploration phase, the selection of different e-nodes or e-classes from the root node may yield different computation expressions (e.g., Extraction #1 and #2 in Figure 2.2(b)). The extracted expressions are equivalent to the input term which is guaranteed by the feature of e-graph. The goal of the extraction phase is to select the best-represented term for each root

Algorithm 1. Equality Saturation.

Input: Starting e-graph G; A set of rewrite rules R; Max e-node limit max_{enode}; Max exploration iterations $max\_explore\_iter$

Output: Explored e-graph G; Nodes selected L_{node}; Objective value $cost$.

$i \leftarrow 0$
while $i < max\_explore\_iter$ **do**
  $G' \leftarrow G$, $i \leftarrow i+1$, $U \leftarrow \emptyset$ // match list
  $U \leftarrow U \bigcup \text{Rule\_Searching}(G, r_j)$ **for** r_j in R
  $G \leftarrow \text{Applying\_Rules}(G, u_j)$ **for** u_j in U
  if $G == G'$ or $num\_node(G') \geq n_{max}$ **then**
    break; // Saturate or max node limitation reached
  end if
end while
$G, L_{node}, cost \leftarrow \text{Extraction}(G)$
return $G, L_{node}, cost$

e-class according to a cost model.

Many extraction methods for equality saturation have been proposed, such as greedy algorithms [111] or ILP solutions [149, 160]. One of the key differences between simplifying DNN arithmetic and previous works of equality saturation (Section 2.7) is that multiple root nodes co-exist during the extraction phase. That is because each weight kernel ($k, k \in \{0, ..., K\}$) in layer L yields different output activation results ($O(x, y, k)$).

In this section, we discuss two strategies to extract low-cost expressions from an expanded e-graph.

Notations. Let i be an e-class where $i = 0, ..., N-1$. Let m be an e-node where $m = 0, .., M-1$. Let $m.op$ denote the operator of the e-node and $m.r_0$ / $m.r_1$ denote the operands of the e-node. Let *root* be the set of root e-classes.

Let e_i and n_m denote a binary integer which indicates if the corresponding e-node i or e-class m is selected or not.

Let $m.r0$ denote the child e-class of operand $r0$ in e-node m, and let $child(i)$ denote the set of child e-nodes of e-class i, i.e., $\{m | m \in child(i)\}$.

Cost Model. In this section, we define our cost model for an e-node m as Eq.(3.2).

$$cost(m) = \begin{cases} C_{add}, & \text{if } m.op = + \\ C_{mult}, & \text{if } m.op = \times \\ 0, & \text{otherwise} \end{cases} \tag{2.3}$$

Here, C_{mult} and C_{add} are constant values. $m.op$ indicates the operator associates with the e-node m. On an Intel i7-8700K CPU, the latency ratio between floating-point (FP) multiplication and FP addition is 3:5. And their reciprocal throughput ratio is 1:2 [42]. Assuming the computation units are fully-pipelined, we choose $C_{mult} = 2$ and $C_{add} = 1$ to make the generated arithmetic expressions more suitable for the deployed hardware. We report complexity for DNN computation as FLOPs, i.e. the sum of additions and multiplications, which are the most commonly used metrics to evaluate the computation complexity of DNN models.

Note that a more complicated and accurate cost model (e.g., dynamic cost) may better characterize the deployed hardware and it is left for future work.

Greedy Extraction Algorithm. We first experiment with a greedy extraction strategy. Q-gym computes the subgraph cost of each e-node using the cost model given in Eq.(3.2). For each e-class, Q-gym selects the e-node with the smallest subgraph cost. Then, the computation expression for each root e-class is determined.

Note that greedy extraction is not guaranteed to extract the graph with the minimum cost. That is because the e-nodes are selected locally in each eclass to minimize the subgraph cost and it fails to consider the potential computation sharing between e-nodes whose subgraphs may be overlapped.

ILP Extraction Algorithms. Alternatively, we can also formulate the selection of e-classes/e-nodes as an integer linear programming problem. The objective of ILP is to optimize the cost of

arithmetic operations:

$$Minimize : f(x) = \sum_{m=0}^{M-1} cost(m) \cdot n_m \tag{2.4}$$

The constraints of the ILP are listed below:

(1) All the root e-classes (*root*) should be included as they represent the final computation results:

$$e_i = 1, \forall i \in \{i | i \in root\} \tag{2.5}$$

(2) Also, we need constraints to say that if an e-class i is included, then at least one of its child e-nodes is included. Otherwise, the term represented by the selected e-class cannot be reached.

$$e_i \leq \sum_{m \in child(i)} n_m \tag{2.6}$$

(3) For each e-node m selected, the child e-class of each operand must be included. This guarantees that the input to the e-node is not empty.

$$n_m \leq e_{m.r0}, \qquad n_m \leq e_{m.r1} \tag{2.7}$$

To recap, the ILP optimization problem can be defined as minimizing Eq.(3.3) which is subject to constraints Eq.(3.4 - 2.7).

2.4.5 Pulsed e-graph Searching

When the size of a weight kernel is large, the e-graph can hardly reach saturation during the exploration phase due to hardware memory limitations. To resolve this issue, we propose an efficient searching algorithm by iteratively conducting *exploration* and *extraction*. Specifically, Q-gym expands the e-graph until the number of e-nodes or exploration steps reaches its predefined limits (i.e., max_{enode}, $max\_explore\_iter$) and extracts the lowest-cost computation expressions using the ILP or greedy methods. Next, Q-gym will explore a new e-graph starting

from the last generated computation expressions as shown in Algorithm 4. Our results show the pulsed searching algorithm can significantly reduce the computation cost throughout iterations (Section 2.6.1).

Algorithm 2. Pulsed e-graph Searching

Input: **Input expressions** $expr_i$**; Set of rewrite rules** R**; Max node count** max_{enode}**; Max searching epochs** $Epochs$

Output: **Output expressions** $expr_o$**; Final e-graph** G'**; Selected node list** L'_{node}

$cost' \leftarrow \infty$
$G \leftarrow$ Initialize_Egraph$(expr_i)$ // Initialize e-graph G
for i in $0, ..., Epochs$ **do**
    $G, L_{node}, cost \leftarrow$ Equality_Saturation(G, R, max_{enode})
    $G \leftarrow Rebuild\_egraph(G, L_{node})$ // Remove all unselected nodes
    if $cost \leq cost'$ **do** $G', cost', L'_{node} \leftarrow G, cost, L_{node}$,
end for
$expr_o \leftarrow$ rebuild_expressions(G', L'_{node}) // rebuild output expressions from e-graph
return $expr_o, G', L'_{node}$

Temporal search space. Q-gym also observes that when $R = S \geq 2$, given a convolutional layer, the input activation overlaps in temporal dimensions when computing $O(x, y, k)$ and $O(x, y+1, k)$. That means computation reuse can also be applied between the computation of different O. As shown in Figure 2.3, if Q-gym searches two continuous convolution operations together, Q-gym can potentially find better computation expressions with lower cost in QNNs. We define the number of convolution operations searched together as timesteps tpr.

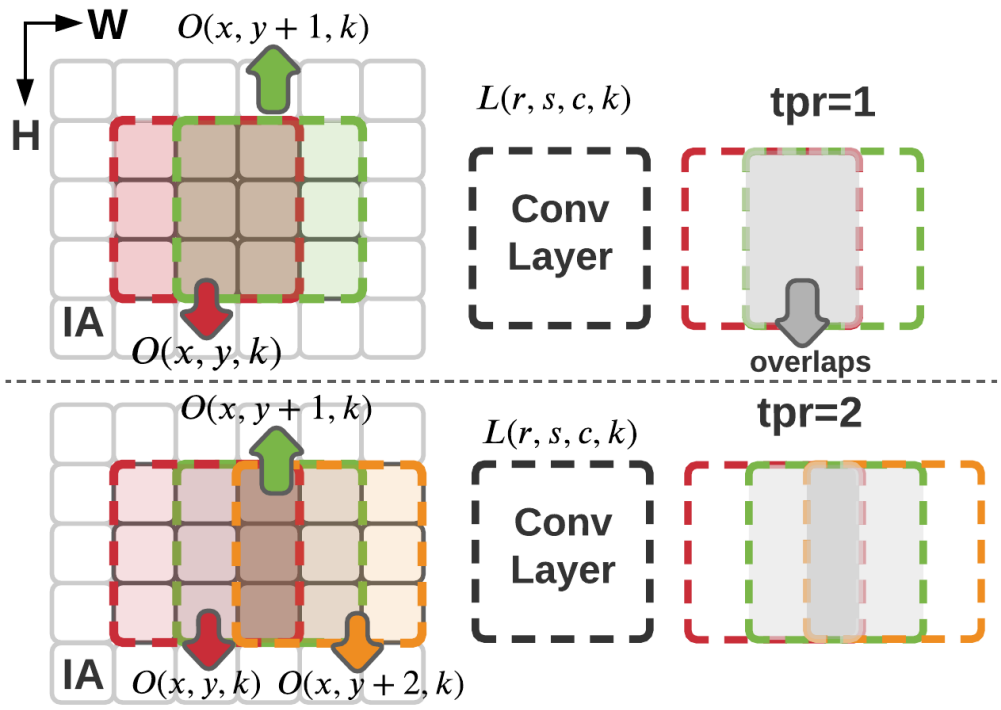


Figure 2.3. An illustration of temporal reuse search space. tpr denotes computations that are explored together. The gray area is the overlapping area of inputs across timesteps. The red / green / orange squares denote the same convolutional layer in different time steps.

2.5 Q-gym's Downstream Tasks

To exploit the efficient computation expressions generated by our compiler, we test Q-gym's performance on various downstream DNN applications.

2.5.1 Accelerating CNN on CPU and GPU Systems

The generated efficient expressions from Q-gym can be used to accelerate CNN inference on CPU/GPU. In this subsection, we detail how to use the generated expressions to yield speedup in DNN inference on CPU/GPU.

Mapping Expressions to Code. As shown in Eq.(2.1), the convolution operation iterates over 6 dimensions (H, W, R, S, C, K). In Q-gym, we unroll the inner for-loop over (R, S, C, K) dimensions. Specifically, the computation over (R, S, C, K) for weight layer L yields K output results (i.e., $O(x, y, k), k \in \{0, ..., K\}$). We replace the unrolled computations with Q-gym's efficient expressions through a common function call. The weight value is hardcoded in the function; so we don't need to load weights during the inference. The output K results from naïve loops over (R, S, C, K) and Q-gym's function call are identical. Note that the convolutions across input dimensions H, W are still repeated in each iteration, so the computation of different output pixels shares the same efficient expressions. No code generation compiler is used.

Combination of Q-gym with Parallelism Methods. By treating the computations along (R, S, C, K) as an atomic function, we can easily adapt the efficient computation expressions with (i) loop tiling, (ii) multithreading, and (iii) vectorization, respectively. In Figure 2.4, we illustrate how to combine the computation of our generated expressions with (i)(ii)(iii).

(i) For loop-tiling, we first split the input activations into tiles with a size of $h_t \times w_t \times C$ to utilize data locality. We exhaustively scan all the possibilities of h_t and select the one that achieves the lowest latency. (ii) For multithreading, we split the w_t into N_{thr} threads to parallelize the computation over w_t. For GPU that allows 1024 threads, we further split the weight kernels along K dimensions into K_{thr} groups (Figure 2.4(b)). Dividing weight kernels along K dimensions can

also reduce the instructions loaded to each core as there is no data dependency across weight kernels. (iii) Vectorization is applied on h_t dimensions to reduce the number of loops along h_t as shown in Figure 2.4. Each addition and multiplication in Q-gym's expressions are compiled into SIMD instructions that operate on different inputs and the same weights. We automate the vectorization by using the *'#pragma omp'* from openMP.

To evaluate Q-gym on CPU, we run two threads per physical core (e.g., 12 threads in total on an Intel i7-8700K processor) to ensure that each core has sufficient instruction-level parallelism (ILP) to fully exploit the available memory bandwidth ($N_{thr} = 12$ and $K_{thr} = 1$). The size of SIMD slots for CPU can be 4/8 (i.e., AVX-2/-512) for FP32. To evaluate Q-gym on GPU, we employ a larger $K_{thr} \times N_{thr}$ to exploit the maximum number of threads allowed on a GPU (1024 threads for NVIDIA 2080 GPU, detailed in Section 2.6.2). We implement the CNN acceleration software for Q-gym's efficient expressions using C and CUDA, by employing the parallelization methods shown in Figure 2.4. The code is compiled with *gcc -O3* on CPU and *nvcc* on GPU. The corresponding evaluation results on GPU and CPU are described in detail in Section 2.6.2.

2.5.2 Accelerating HE for DNNs

We identify that the low-cost computation arithmetic from Q-gym's compiler significantly reduces the evaluation time of DNN under homomorphic encryption (HE) as discussed in Section 2.2. The generated computation dataflow can be easily combined with different HE libraries (i.e., HELib [3], SEAL [135]) by simply replacing the evaluation function f with the expressions compiled from Q-gym without any modifications to the encryption protocol. Note that Q-gym does not increase the depth of multiplication in HE; so it will not affect the correctness of the output result. We evaluate Q-gym's efficient expressions using two popular HE protocols (BGV [15] / BFV [41]). We also verify the correctness of the Q-gym's expressions using HELib/SEAL. HELib/SEAL with BGV/BFV schemes are also used to implement the CryptoNet/FastCryptoNet, respectively.

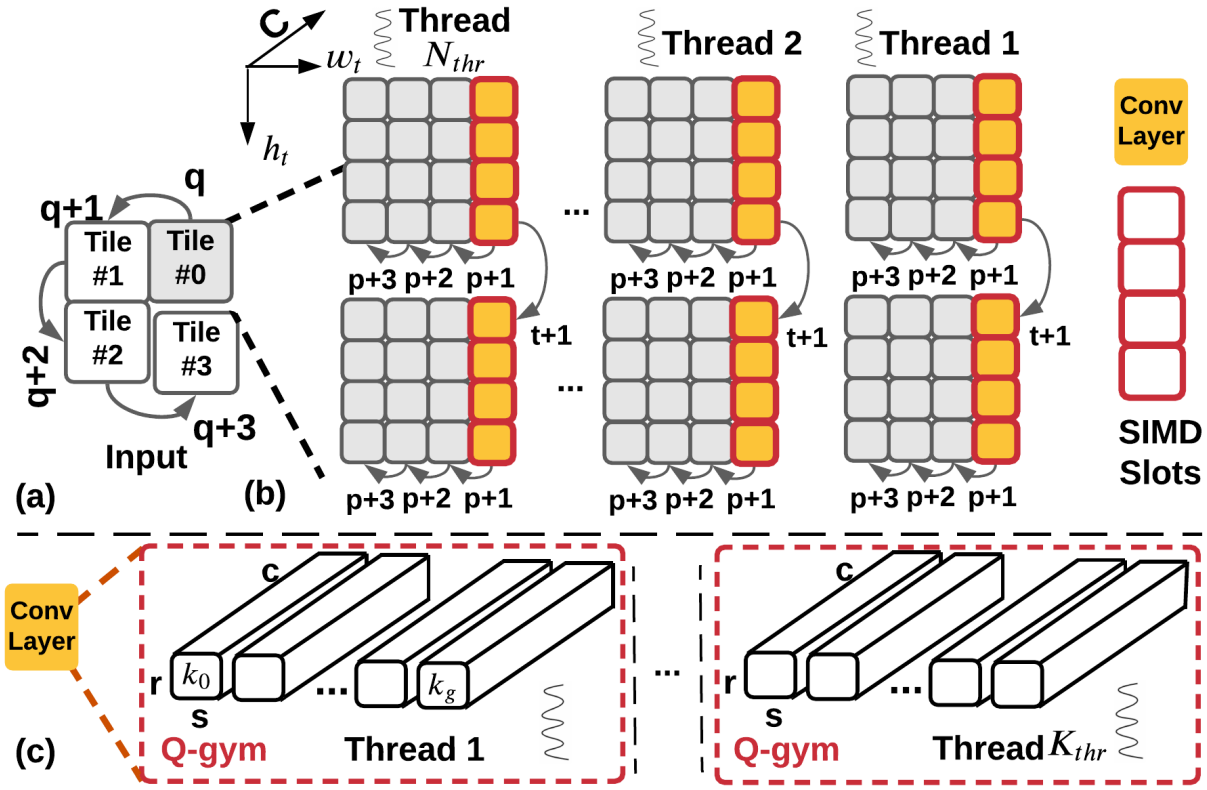


Figure 2.4. The parallelization mechanism in Q-gym with loop tiling, multithreading, and vectorization. (a) We split the input into small tiles for better data locality. q is an iterator over input tiles. (b) We apply SIMD vectorization along h_t of each tile to reduce the number of loops. We also split the tile along w_t dimension into N_{thr} groups and (c) K kernels into K_{thr} groups. Each group with is handled by different threads. $N_{thr} \times K_{thr}$ is the total number of threads. p, t are iterator over w_t and h_t. For simplicity, we set $r = s = 1$ in this figure.

Note that the runtime of DNN inference under HE is linear to the number of HE operations. Specifically, there are four types of HE operations in DNN inference: (i) plaintext (PT)-ciphertext (CT) addition, (ii) ciphertext-ciphertext addition, (iii) plaintext-ciphertext multiplication, and (iv) ciphertext-ciphertext multiplication. Because the wall-clock time for executing different types of HE operations is library-dependent and is highly relevant to the other settings (e.g., key size, machine settings), in this section, we use the number of HE operations (HOPs) for comparison. This metric is commonly used for other privacy-preserving DNN methods, such as FastCryptoNet [27].

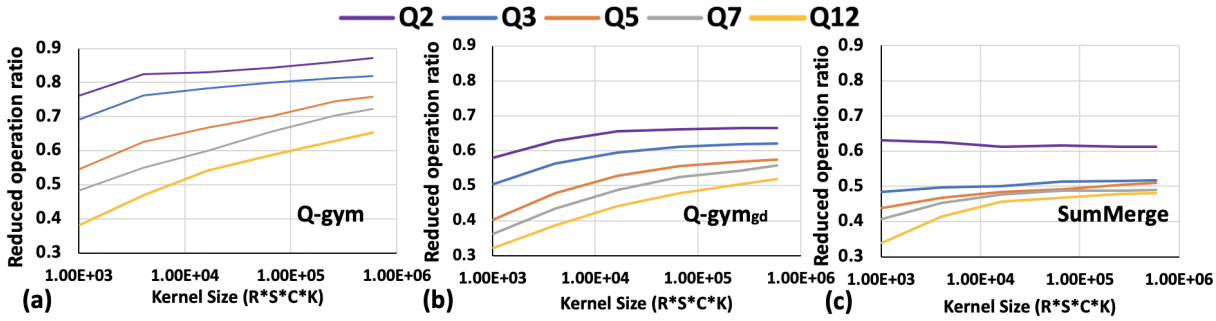


Figure 2.5. Comparison of reduced operations ($-ops$) between (a) Q-gym (b) Q-gym$_{gd}$ (c) SumMerge over different quantization schemes (Q) and layer sizes ($R \cdot S \cdot C \cdot K$).

2.6 Evaluation

In this section, we evaluate the performance of our compiler design in reducing QNN computations (Section 2.6.1) and the performance of Q-gym in accelerating various DNN applications (Section 2.6.2).

2.6.1 Algorithm Analysis

Experimental Setup. We implement the algorithm of Q-gym in Rust from egg [162], an open-source equality saturation library. During the e-graph extraction phase, we use Gurobi [1] as the ILP solver. We also reimplemented SumMerge in Python with loop parallelism as the baseline. The compilation time is tested on Intel Core i7-8700K CPU with 6 physical cores. This

Table 2.3. CPU/GPU configurations.

CPU	Intel i7-8700K, 6 physical Cores, 3.7 GHz, 1 socket
GPU	NVIDIA 2080, 8GB main memory, 1024 maximum threads
L1 Cache	32KiB 8-way I$, 32KiB 8-way D$, private
L2 Cache	256KiB, 16-way, private
L3 Cache	12MiB, 11-way, shared
TLB	L1D 4-way 64 entries, L1I 8-way 128 entries STLB 12-way 1536 entries
DRAM	DDR4, 32GB, 2666MHz, 2 sockets, 6 channels per socket
Kernel	Linux 5.4.0
Software	GCC 7.1, PyTorch 1.4.0+cuDNN, CUDA 11.2, Rust 1.56.1

machine has 32/32 KiB of L1 instruction/data cache, 256 KiB of L2 cache per core, and 12 MiB of shared L3 cache (Detailed in Table 2.3).

To measure the performance of Q-gym in reducing computations, we use the ratio between the number of reduced FLOPs and the total number of FLOPs ($-ops$) as the evaluation metric. For comparison purposes, we conduct experiments with Q-gym that use the greedy method during extraction (denoted as Q-gym$_{gd}$). Also, Q-gym without the pulsed searching algorithm is denoted as Q-gym$_{-p}$ (i.e., $max_{epoch} = 1$).

Without specification, we set max_{enode} = 10^7, epochs max_{epoch} to 10, $max\_explore\_iter$ = 8, and set the temporal reuse steps (tpr) to 0 for Q-gym. For ILP extraction time limitation, we set it to $\sqrt{R \cdot S \cdot C \cdot K}$ seconds which is relative to the number of weight kernels in the convolutional layer.

Comparison of Reduced Operations. Figure 2.5 compares the reduction of FLOPs between Q-gym, Q-gym$_{gd}$, and SumMerge. We evaluate the algorithms on QNNs where the weights are

generated synthetically using a uniform distribution. We assume none of the weights are 0 in the synthesized layers.

The result shows that Q-gym achieves significant computation reductions compared to SumMerge on quantized DNN with different Q. On average, Q-gym shows 19.1% / 15.5% more computation reduction than SumMerge in Figure 2.5. Q-gym also achieves a more substantial operation reduction compared to Q-gym$_{gd}$. As discussed in Section 2.4.4, using an ILP solver can find a better solution with a lower computation cost compared to a greedy heuristic.

Q-gym$_{gd}$ can also outperform SumMerge. On average, Q-gym$_{gd}$ shows 3.6% more computation reduction across all the kernels tested in Figure 2.5 (2.3% / 8.3% on average for $Q = 2$ / $Q = 3$). This is because e-graph exploration covers a larger search space compared to searching the common expressions between activation groups.

Compilation Time of Q-gym's Algorithm. We evaluate the time for Q-gym to compile a given weight layer into efficient computation expressions. Figure 2.6 shows the compilation time of Q-gym. With the increasing weight dimension and unique weights (Q), SumMerge takes a much longer compilation time than Q-gym. That is because SumMerge checks the overlapping of terms between all activation groups for computing the 'maxscore' (Section. 2.3). Also, the 'maxscore' is computed many times until no overlapping between activation groups is found.

With the increase of Q, the compilation time of Q-gym does not change a lot due to the e-node/explore step limitation (max_{enode}, $max\_explore\_iter$) and max epochs (max_epochs) for pulsed searching. To apply Q-gym in runtime compilation techniques, the users can choose Q-gym$_{gd}$ which is much faster compared to Q-gym and SumMerge.

Sensitivity to the Number of Unique Weights (Q) and Layer Size. With a growing number of weights, the performance of Q-gym also increases (Figure 2.5). When $Q = 12$, the computation reduction ratio of Q-gym increases from 38.3% to 62.9% with the growing size of weights. For SumMerge, the performance eventually converges at $\sim$50%. This means the greedy heuristic cannot fully exploit the computation reuse.

With the growing number of unique weights (Q), Q-gym always shows more compu-

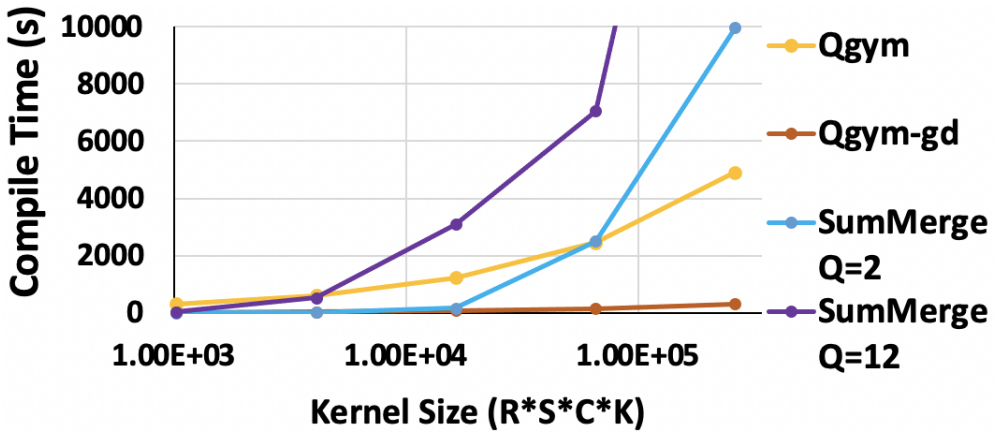


Figure 2.6. Compilation time for different weight sizes using SumMerge, Q-gym and Q-gym$_{gd}$.

tation reduction compared to SumMerge. Specifically, for $Q = 2$ / $Q = 3$ / $Q = 12$, Q-gym shows 21.3% / 27.6% / 11.2% more computation reduction compared to SumMerge. Note that SumMerge's performance decreases with the increase of Q. This is because the activation groups in SumMerge are factorized into smaller sets when Q is large, offering fewer opportunities for computation reuse.

Sensitivity to Temporal Reuse Steps (tpr). Figure 2.7 shows a sensitivity analysis of tpr on different QNN layers. With larger R, more inputs are overlapped across time steps and achieve lower computation cost when $tpr \geq 1$. For weight layers where (R,S) is $(3,3),(5,5),(7,7)$, Q-gym where $tpr = 3$ shows 4.7, 6.0%, and 6.2% more computation reduction compares to Q-gym where $tpr = 0$. With the increasing tpr, the overlapping area of input activations also increases (Figure 2.3).

Comparison between Q-gym and Winograd. We also compare Q-gym with temporal reuse against Winograd, a commonly used technique that leverages fast FFT to reduce computation operations in convolutional layers. The downside of Winograd is that the transformation overhead is not negligible, i.e., the storage overhead of transformation matrices and the computation of input, weight transformations, and inversions. Also, Winograd has a theoretical boundary of 75% computation reduction (factor of 4). In this comparison, we did not take into consideration

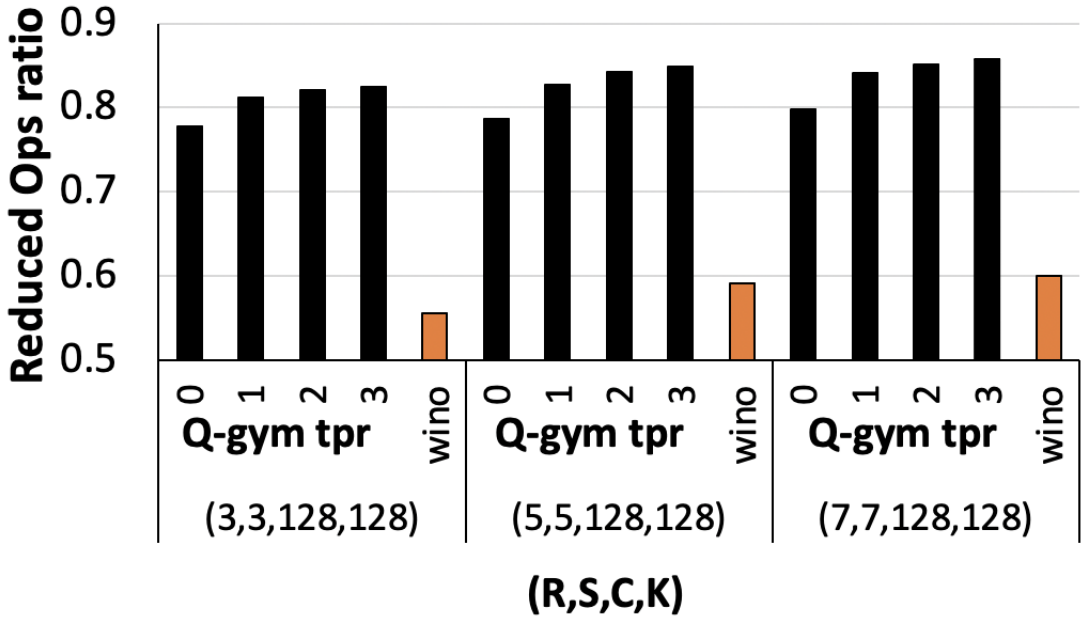


Figure 2.7. Sensitivity of tpr over the reduction of operations ($-ops$) when $Q = 3$. Comparison between Q-gym with temporal reuse to Winograd (denoted as 'wino').
We assume the input tile size ((H,W)) for Winograd is $(4,4)$, $(12,12)$, $(25,25)$ for weights (R,S) $(3,3)$,$(5,5)$,$(7,7)$, respectively. The corresponding α' are 2.25 / 10.51 / 19.61 in Winograd [87].

the extra computation for input, weight transformation, and inversion for Winograd. Thus, the baseline is stronger than what happens in practice.

On average, Q-gym shows 24.2% more computation reduction compared to Winograd as shown in Figure 2.7. Note that Q-gym can be combined with Winograd and we leave it as future work.

Analysis of Q-gym on group convolutions. We also verify if Q-gym is still working in group convolution layers [178, 131, 148]. Traditional convolution layers have a weight dimension (R,S,C,K) as given in Eq.(2.1). With group convolution where the grouping factor is g, the channel of each weight kernel would be $C_g = C/g$. The K kernels will be separated into g groups and each group has $K_g = K/g$ weight kernels, i.e., each group has a dimension of (R,S,C_g,K_g). When $g = C$, the convolution layer will become a *depth-wise convolution layer*. During group convolution, the input activation can be reshaped into (H,W,C_g,L_g). Each group of kernels (R,S,C_g,K_g) handles different groups of the input activations (R,S,C_g) in the same fashion given

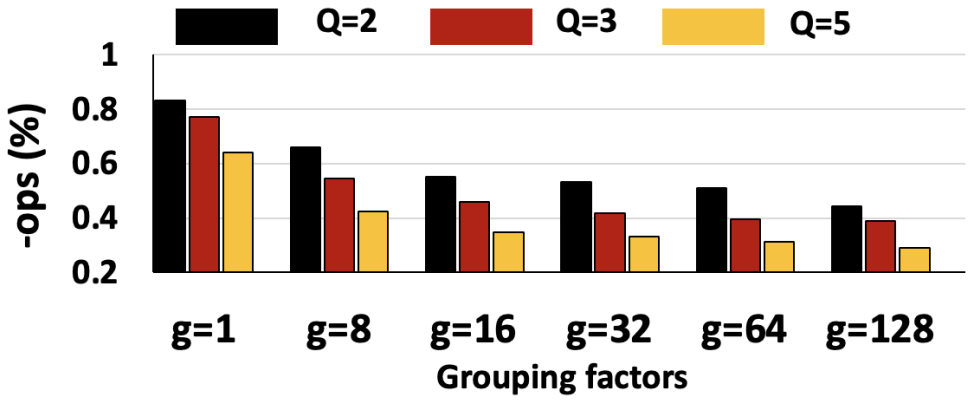


Figure 2.8. Sensitivity of operation reductions ($-ops$) over grouping factors g. The tested kernel size is $(3,3,128,128)$ and we set $tpr = 0$.

in Eq.(2.1). As such, Q-gym can still handle group convolutions when $R \times S \times C_g \times K_g > 1$. But with the increase of g, the search space decreases, giving Q-gym less opportunity to reuse computations. As shown in Figure 2.8, when $Q = 3$, the operation reduction drops from 77.1 % of traditional convolution to 38.9 % for depth-wise convolution layers.

Note that most of the depth-wise or group convolutions are implemented together with traditional convolutional layers [178, 131, 148]. Also, these models are typically not quantized due to a large accuracy drop.

Effectiveness of Pulse Searching Algorithm. Figure 2.9 shows the operation reductions over different epochs during Q-gym's pulse searching. On average, Q-gym with pulsed e-graph searching can reduce the computation cost by 9.9% compared to Q-gym$_{-p}$. For small kernels, e.g., $(1,1,32,32)$, the computation reduction of pulse searching is limited due to the relatively small search space. For large kernels, pulse searching shows substantial performance improvement over Q-gym$_{-p}$. Note that from epochs 5 to 7, the number of operations is still reducing across different test cases. However, overall the number of reduced FLOPs becomes negligible compared to the total FLOPs.

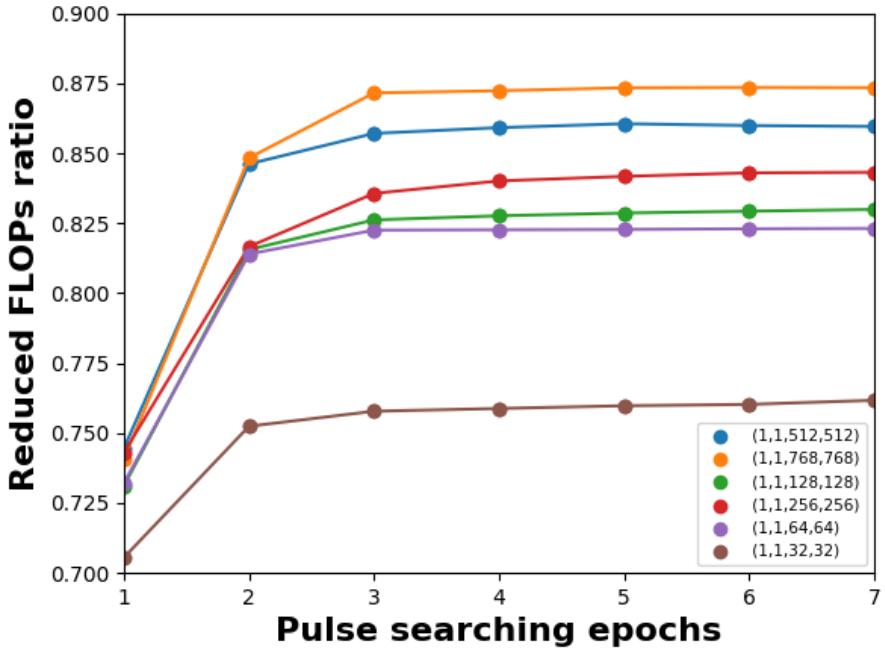


Figure 2.9. Computation reductions ($-ops$) over the pulsed searching epochs across different convolutional layers (R,S,C,K).

Table 2.4. Comparison between Q-gym and SumMerge/OneDNN on CPU performance. ♭TTQ is a sparse and quantized model (one of the three values is 0. ♮ VGG-small is a derivative architecture based on VGG [141]. ♯ '-Ops Gap' denotes the gap of computation reduction ratio ($-ops$) between Q-gym and SumMerge. The models can be found in the repository [173, 182].

Description				Accuracy		v.s. SumMerge [117]			v.s. OneDNN [2]	v.s. PyTorch [112]
DNN Arch	Q	Dataset	Quantized method	Full	Quantized	♯ -Ops Gap	(%)	CPU MT Speedup	CPU MT Speedup	GPU MT Speedup
AlexNet [84]	2	ImageNet [36]	BWN [124]	59.7	55.7	27.1		1.83	1.76	1.03
AlexNet	3	ImageNet	♭ TTQ [184]	59.7	55.2	16.7		1.45	1.84	1.42
ResNet-20 [57]	3	CIFAR-10 [82]	ProxQuant [6]	91.9	91.3	27.0		1.21	2.31	1.86
♮VGG-small[18]	4	CIFAR-10	LQ-Nets	93.8	93.5	21.9		1.40	1.32	0.95
ResNet-18	2	CIFAR-100 [83]	LS-1 [116]	77.8	75.8	27.6		1.55	1.81	1.56
ResNet-18	3	CIFAR-100	LS-T [116]	77.8	76.5	29.8		1.52	1.59	1.20

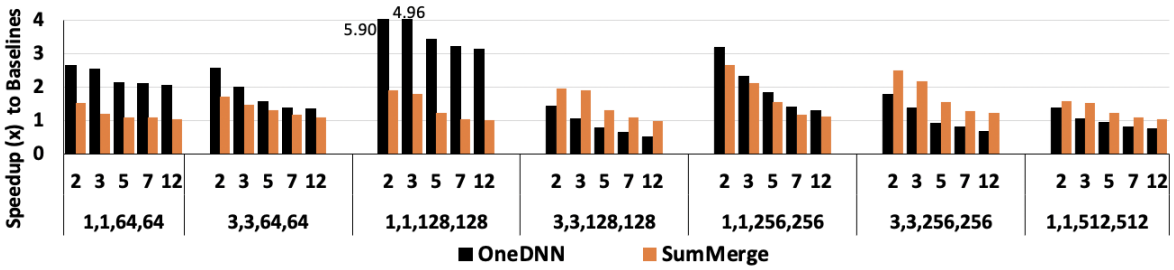


Figure 2.10. Per-layer speedup (higher is better) comparison on CPU. The tuple (R,S,C,K) denotes the size of different convolution layers. For OneDNN implementation, we use the official performance profiling tool.

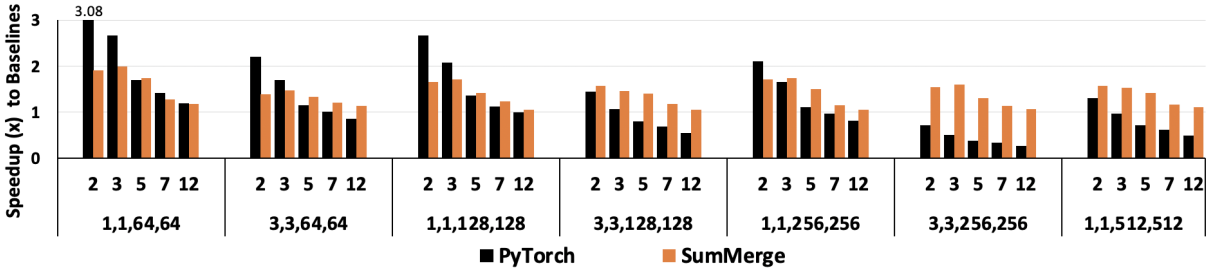


Figure 2.11. Per-layer speedup (higher is better) comparison on GPU.

2.6.2 Evaluation of Q-gym's Downstream Tasks

Experimental Setup. We evaluate the performance of Q-gym on an Intel Core i7-8700K CPU (The same machine as Section 2.6.1) and an NVIDIA 2080 GPU (1024 max threads), respectively. For CPU and GPU baselines, we choose to compare against the expressions generated from SumMerge. We also compare against the state-of-the-art DNN compiler (OneDNN [2])) for Intel CPU and Pytorch GPU (v1.4.1) [112] which has a carefully designed cuDNN/CUDA (v11.2) back-end for performance purposes. All the weights and input activations are 32-bit floating-point numbers. For all comparisons, we set the batch size to be 1 to accommodate the common configurations used in real-time systems. The implementation methods of Q-gym for CPU/GPU are described in Section 2.5.

Performance of Q-gym on CPU. Figure 2.10 shows the per-layer speedup with different Qs on the CPU. All experiments assume an input of dimensions of $H = W = 48$. We enable multithreading (MT) for both Q-gym and OneDNN. For MT implementation in Q-gym, we set $N_{thr} = 12$ and $K_{thr} = 1$ across all layers tested to maximize CPU resource utilization. For OneDNN settings, we scan the number of threads from 1 to 12 for each layer and select the best-performing setting. For SumMerge, we apply the same MT implementation as Q-gym (Section 2.5) with the computation expressions generated from SumMerge. For vectorization, we unroll the loop along H dimensions (unrolled factor = 8).

When $Q \leq 3$, Q-gym shows 2.56× / 1.83× speedup compared to OneDNN and SumMerge on average across different convolution layers. Note that $Q \leq 3$ are common settings for QNNs (Table 2.4). We observe that when the Q gets larger, the speedup of Q-gym over SumMerge decreases due to the relatively lower computation reduction of Q-gym in large Q. On real-world DNN models shown in Table 2.4, Q-gym achieves 1.49× / 1.77× speedup over SumMerge / OneDNN with an average accuracy loss of 2.12% relative to the full precision models.

We also compare the performance of Q-gym on CPU to its theoretical peak as shown

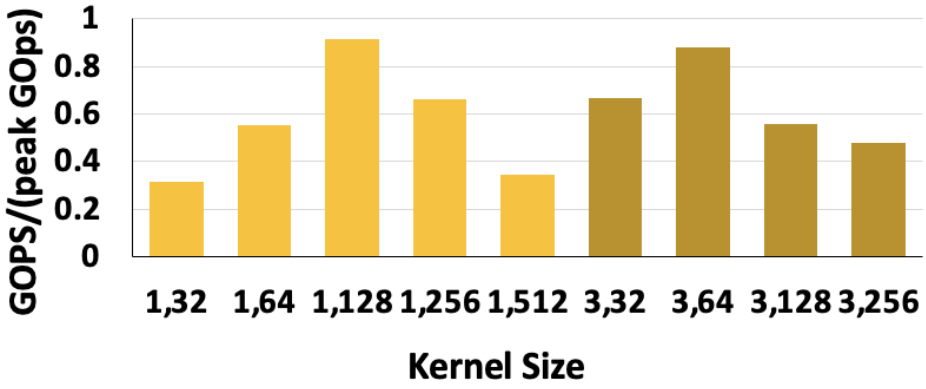


Figure 2.12. Per-layer GOPS and peak GOPS on CPU across different convolutional layers. The tuple (R,C) denotes the size of the kernel (R,S,C,K) where $R = S$, $C = K$.

in Figure 2.12. Specifically, we assume the two VPUs (SIMD width is 8) in each core are fully pipelined running on 3.7GHz. All 6 physical cores are enabled. As shown in Figure 2.12, when the kernel size is small, the performance ratio is low. That is because the threads' creation and termination overhead are relatively large compared to the computation overhead. For large kernels, the CPU utilization is also decreasing. This is because large convolution kernels yield a large number of computation expressions and the data locality between expressions is getting worse.

Performance of Q-gym on GPU. Figure 2.11 shows the per-layer speedup on GPU over PyTorch and SumMerge. Regarding the settings of Q-gym's parallelization method (Figure 2.4), we choose $K_{thr} = 8192/C$ to limit the size of instructions loaded to the GPU cache. We also set $N_{thr} = 1024/K_{thr}$ to maximize the usage of GPU resources. All runs assume an input of dimensions $H = W = 112$.

Q-gym shows an average speedup of $1.92\times$ / $1.64\times$ compared to PyTorch when $Q = 2$ / $Q = 3$. Also, Q-gym shows an average speedup of $1.63\times$ when $Q \leq 3$ over the computation expression from SumMerge. With the increasing size of weight layers, the performance of Q-gym drops faster than PyTorch due to the larger instruction size and memory footprint. Also, for the convolution layer where H and W are small, Q-gym parallelizes on the K dimension to

maximize the usage of GPU resources. This will reduce the search space of Q-gym's compilation phase and reduce the opportunity for computation reuse.

Performance of Q-gym for HE. As mentioned in Section 2.5.2, we compare DNN inference under HE using the homomorphic operations (HOPs) following previous works [27]. Due to the expensive computation cost of HOP on real hardware, the inference time of HE applications is linear to the number of HOPs regardless of the hardware or software parallelization schemes.

For the baseline comparison, we choose CryptoNet [48], a dense DNN model that is trained on MNIST [88]. FastCryptoNet [27], a follow-up work of CryptoNet that leverages sparsity to bypass HOPs. We use a trained dense QNN and a trained sparse QNN to compare with the baselines separately. The model architectures are the same as FastCryptoNet (a slight variant of CryptoNet). We apply Q-gym to compile the models for HOPs comparison.

As is shown in Table 2.5 and 2.6, Q-gym achieves 59.5% and 22.9% HOPs reduction relative to CryptoNet and FastCryptoNet. Meanwhile, the models trained using INQ achieve almost the same accuracy (-0.11%/-0.02%) compared to CryptoNet / FastCryptoNet. Besides, Q-gym reduces the PT-CT multiplication by 10.8× / 1.87× which is the computation bottleneck of DNN inference under HE.

Deploying Q-gym on Small Devices. During code generation, Q-gym unrolls the inner convolution loops and replaces them using efficient computation expressions. As such, the compiled binary size is larger than the one using naive loops. For small architectures (e.g., mobile devices) with a small instruction cache, the large number of instructions may decrease the performance of Q-gym. On those devices, Q-gym can be employed using an alternative option by leaving the computation expressions in the "interpreted form", i.e., we load the expressions during inference and compute the output accordingly, which is the way implemented in SumMerge [117]. Q-gym would then still be more efficient than the "interpreted" SumMerge due to better performance in reducing operations. On our experiment architectures, Q-gym's deploying method is better and we use the same way to evaluate SumMerge in this section.

Table 2.5. A breakdown of HOPs for each layer between Q-gym and CryptoNet. The dense model for Q-gym uses 2-bit INQ [180] where $Q = 4$. The model architecture (same as FastCryptoNet) for Q-gym is a slight variant of CryptoNet. We set $tpr = 0$ for all layers' compilation. Fully-connected (FC) layers can be treated as special convolutional layers (conv) where R, S, H, W are 1. 'Act' denotes activation layers.

CryptoNet					
Layer	Hops	PT-CT Adds	CT-CT Adds	PT-CT Mults	CT-CT Mults
Conv-1	42,757	845	20,956	20,956	-
Act-1	845	-	-	-	845
Pool-1	6,845	-	6,845	-	-
Conv-2	309,905	1,250	154,350	154,350	-
Pool-2	8450	-	8450	-	-
FC-1	241192	100	120546	120546	-
Act-2	100	-	-	-	100
Fc-2	1990	10	990	990	-
Total	612,129	2,205	312,137	296,842	945
Accuracy	99.17				
Q-gym					
Conv-1	25,350	1,690	15,717	7,943	-
Act-1	5,070	845	1,690	1,690	845
Pool-1	6,845	-	6,845	-	-
Conv-2	118,975	1,250	105,225	12,500	-
Pool-2	-	-	8450	-	-
FC-1	82097	100	76997	5000	-
Act-2	600	100	200	200	100
Fc-2	477	10	427	40	-
Total	247,864	3,995	215,551	27,373	945
Accuracy	99.06				

2.7 Related Work

This section discusses previous works related to our study.

Quantized Neural Networks. Neural network quantization [180, 184, 173] is a promising technique to compress and accelerate DNNs. The reduced bit width enables low-bit width multiplication [137] to speed up the inference. Jung et al. introduced parameterized quantization intervals and optimized them to minimize task loss [75]. Han et al. used k-means clustering as a method of quantization to share weights [56]. Zhou et al. proposed a rule-based non-uniform quantization by leveraging a logarithmic distribution [180]. [173, 184] optimize the quantization clipping range during model training. Beyond the computer vision tasks, quantization can also

Table 2.6. Comparison between Q-gym and FastCryptoNet. As FastCryptoNet is a sparse model $sp = 6.63\%$ and applied 2-bit INQ quantization [180], Q-gym uses a sparse and quantized model ($Q = 4$) for evaluation accordingly. Sparse ratio sp is set to be the same as Fast CryptoNet.

FastCryptoNet					
Layer (sp)	Hops	PT-CT Adds	CT-CT Adds	PT-CT Mults	CT-CT Mults
Conv-1	8,619	1,690	3,042	3,887	-
Act-1	5,070	845	1,690	1,690	845
Pool-1	6,845	-	6,845	-	-
Conv-2	22,950	1,250	10,850	10,850	-
Pool-2	8450	-	8450	-	-
FC-1	14,354	100	7,077	7,177	-
Act-2	600	100	200	200	100
Fc-2	306	10	148	148	-
Total	67,194	3,995	38,302	23,932	945
Accuracy	98.73				
Q-gym					
Conv-1	6,760	1,690	2,704	2,366	-
Act-1	5,070	845	1,690	1,690	845
Pool-1	6,845	-	6,845	-	-
Conv-2	18,150	1250	10550	6350	-
Pool-2	8450	-	8450	-	-
FC-1	5,724	100	3383	2241	-
Act-2	600	100	200	200	100
Fc-2	192	10	131	51	-
Total	51,791	3,995	33,953	12,798	945
Accuracy	98.71				

be applied to BERT [176, 138] for NLP tasks. Q-gym can be applied to all these quantization models to reduce the computations of QNNs.

Equality Saturation Applications. The key idea of Q-gym is equality saturation [149, 142]. It has also been widely used in various domains, such as simplifying CAD design, rewriting DNN architectures, improving the accuracy of floating-point expressions and doing semantic code search [167, 111, 118, 107]. In this section, we contribute to the iterative searching algorithm and elaborated search space to better reduce the computation in DNN. This is also the first work that applies equality saturation to accelerate homomorphic encryption for quantized DNNs inference.

Acceleration of Homomorphic Encryptions for DNNs. Since the first Fully HE scheme was proposed by Gentry et al. [47]. Many acceleration methods for FHE have been proposed, such as Leveled Somewhat HE (SWHE) [15, 7, 14]. Many advances [41, 76, 13] leverage SWHE to do privacy-preserving DNN inference. CryptoNet [48] is the first work to use SWHE for DNN inference. FastCryptoNet [27] leverages DNN model sparsity to accelerate CryptoNet. Many other works have aimed at improving the computation efficiency of non-linear layers [101] or computing SWHE in a SIMD fashion [76]. These methods are orthogonal to the computation reduction generated from Q-gym and can be combined to further improve the efficiency of HE DNN inference. Note that Q-gym does not tamper with the security of HE. The input is kept private all the time and no information is leaked.

2.8 Conclusion

In this chapter, we propose Q-gym, a DNN framework that accelerates quantized DNNs by exploiting the weight repetition characteristic. Q-gym proposes a compiler and a set of acceleration schemes for various DNN applications. For the compiler, Q-gym employs the idea of equality saturation by representing the DNN computation into an e-graph and applies a set of rewrite rules to explore equivalent expressions. After the exploration, we extract the

lowest-cost computation expressions from the e-graph by formulating the selection of nodes in the e-graph into an integer linear programming problem. By iteratively conducting exploration and extraction and leveraging a temporal search space, we can further reduce the computation operations compared to previous works.

Leveraging the reduced computation, Q-gym can accelerate a set of DNN applications. We build QNN inference kernels on CPU and GPU with carefully designed parallelization schemes and combine the efficient expressions from Q-gym with multi-threading, vectorization, and loop tiling. Q-gym also proposes to combine the reduced expressions into DNN inference flow under homomorphic encryption. Experiments show significant speedups and operation reductions in those applications compared to the state-of-the-art methods.

Acknowledgement. Chapter 2, in part, contains a re-organized reprint of the material as it appears in International Conference on Parallel Architectures and Compilation Techniques (PACT) 2022. Cheng Fu; Hanxian Huang; Bram Wasti; Chris Cummins; Riyadh Baghdadi; Kim Hazelwood; Yuandong Tian; Jishen Zhao; Hugh Leather. The dissertation author was the primary investigator and author of this paper.

Chapter 3

A Hardware-friendly Transfer Learning Framework Exploiting Computation and Parameter Sharing

3.1 Introduction

Fine-tuning from pre-trained transformers [155] has become the *de-facto* method for many NLP tasks, with performance breakthrough in various natural language understanding benchmarks [38, 86, 97, 73, 168]. Yet, the growing number of different NLP tasks arriving in stream makes this approach hard to integrate into real-world commercial products. The key bottlenecks lie in both **computation** and **storage** constraints. In particular, with conventional fine-tuning methods [64, 157], both single input processing latency and storage requirement grow linearly to the number of sub-tasks. This incurs an impractical computation, power, and storage overhead for a commercial product.

Both computation and storage constraints are critical to fine-tuning tasks. On the one hand, without much sacrifice in the quality of service, cloud computing vendors care more about the computation constraint to further improve the quality. The storage overhead can potentially be resolved by the advances in memory and storage technologies [70, 30, 31], which enable the low-cost and large-capacity data storage. On the other hand, in memory-limited devices (e.g., mobile), both constraints are critical to user experience. Most prior works focus on reducing

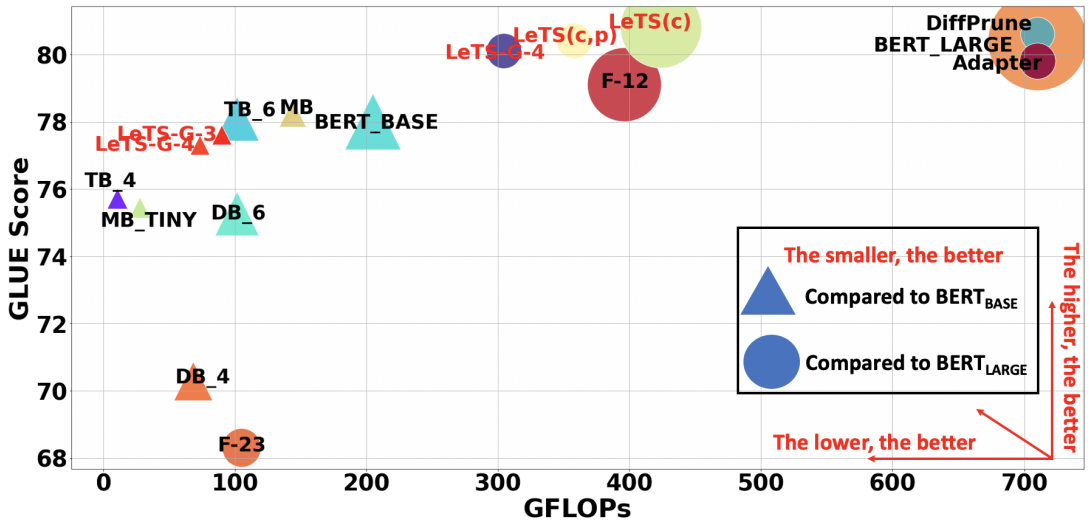


Figure 3.1. Roadmap of GLUE score v.s. total operations and parameters. The area of a point is proportional to the parameter size. *F-n* denotes freezing bottom *n* layers. *MB, TB, DB* represent MobileBERT, TinyBERT, DistilBERT respectively. LeTS outperforms other parameter-sharing methods in terms of computation and parameter efficiency. LeTS is orthogonal to model compression techniques (e.g. MB/TB/DB) as LeTS does not modify the pre-trained model for fine-tuning.

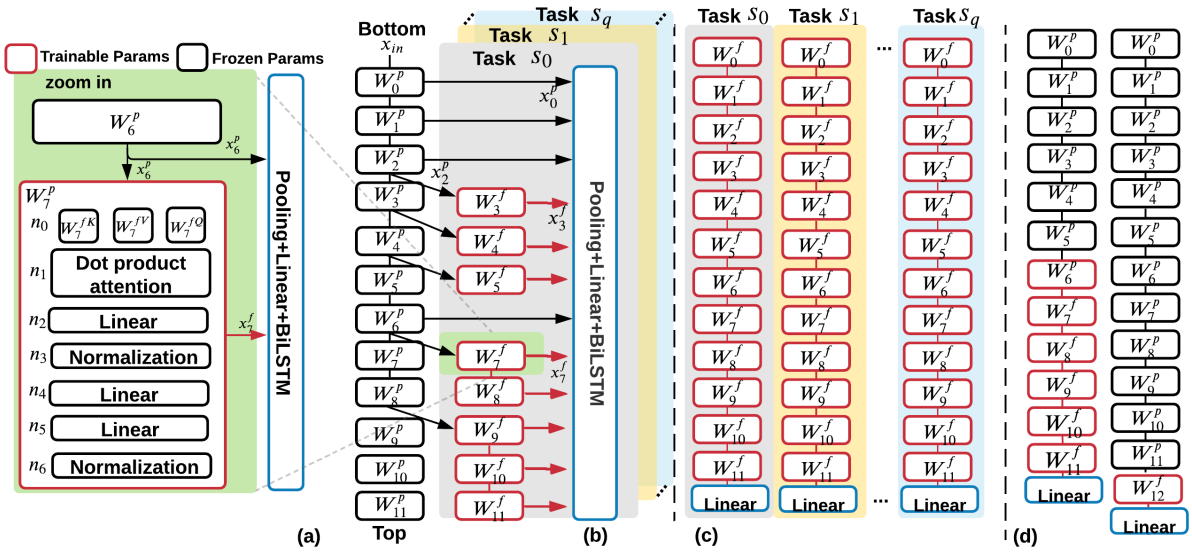


Figure 3.2. (a) Zoom-in of a single self-attention layer. (b) An example searched model on $\text{BERT}_{\text{BASE}}$. x_j^p can be reused by all the sub-tasks. (c) Traditional fine-tuning paradigm on the sub-tasks. (d) Baseline models, i.e., freezing bottom 6 layers / Appending 1 layer on top of the pre-trained model.

the storage constrained across multiple sub-tasks by leveraging parameter-sharing. Multi-task learning (MTL) trains all the sub-tasks together [96, 28, 143]. However, it requires access to all the sub-tasks at the design time. Furthermore, MTL is not scalable with the increasing number of sub-tasks, as it is hard to balance the performance of multiple tasks and solve them equally well [143].

Recently, Adapter [63] considers the new tasks in the fashion of arriving in stream which is more scalable compared to MTL. It proposes to add a task-specific building block between each attention layer and freeze the other parameters during fine-tuning. Recent works propose a differentiable pruning method [53] that achieves better results than Adapter.

Unfortunately, all the aforementioned parameter-efficient efforts do not address computation bottleneck in multi-task inference, because tuning the bottom layers will influence the computation results in the downstream layers. As such, re-computation is required.

To mitigate both the computation and storage burdens in multi-task evaluation, we propose **Le**arn-**t**o-**S**hare (*LeTS*), a new transfer-learning framework that exploits both computation- and parameter-sharing to reduce computation and storage demands, while keeping high performances on sub-tasks. The key contributions are as follows:

(i) We propose a new fine-tuning architecture design space (Figure 3.4). The output of each self-attention layer will be aggregated at the end using a pooling layer and a bidirectional LSTM [68] (Bi-LSTM) to obtain the final classification result. In this way, modifications on the bottom layers do not influence the downstream computation. This enables concurrent execution inside the transformer. Many computations can be bypassed when the Bi-LSTM uses the attentions that are already computed as input. Also, we identify that even more computations can be reduced by converting matrix-matrix multiplications to matrix-vector multiplications (Sec. 3.3.2).

(ii) We design a differentiable neural architecture search (NAS) algorithm to find an optimal fine-tuning architecture for a sub-task. Specifically, NAS selects the input to each attention layer and the final pooling layer. When a computed result is selected as the layers' input,

we can bypass many computations to achieve computation sharing. A new computation-aware loss function for our search space is proposed to search models that can reduce computation and preserve task accuracy.

(iii) We treat the obtained **f**ine-tuning model weights as the sum of **p**re-trained weights and weight difference ($\boldsymbol{\delta}$): $W^{\mathbf{f}} = W^{\mathbf{p}} + W^{\delta}$, and propose a novel early-stage pruning method to obtain $W^{\boldsymbol{\delta}}$. A weight mask to represent pruning is generated for W^{δ} at the beginning of the fine-tuning. Rather than randomly initialized, W^{δ} is initialized with task-specific gradient accumulation to get a robust weight mask.

(iv) We systematically integrate (ii) and (iii) to produce fine-tuning models with high task performance and low computation and storage costs. During NAS, a generated mask from (ii) on the trainable parameters can better characterize the model performance. Also, during the online prototyping, when the input and output of a given linear layer are already computed, the computation can be reduced into a sparse-matrix multiplication by leveraging the sparsity produced from (iii).

Our framework produces efficient fine-tuning language models for different computing environments. Extensive experiments show that LeTS reduces computation cost by a large margin while achieving a *competitive sub-task accuracy*. More specifically, for computing and storage-restricted platforms, LeTS yields 49.5% computation reduction by adding only 1.4% extra parameters per task while preserving a high task accuracy of the fine-tuned BERT [38] on GLUE benchmarks. For a computing environment with a low-cost storage budget, LeTS can achieve 40.2% computation reduction with no accuracy loss. Moreover, LeTS saves more computation per task with more sub-tasks. For $\text{BERT}_{\text{BASE}}$, LeTS requires 7.2 GFLOPs[1] for every newly added task compared to 22.5 GFLOPs of a fine-tuned $\text{BERT}_{\text{BASE}}$.

LeTS is the first framework that considers both computation and storage efficiency in fine-tuning for multi-task NLP. Our work can be combined with model compression techniques [86, 132] to enable agile and efficient NLP evaluation.

[1] 1 GFLOPs = 1 billion floating-point operations

3.2 LeTS: Overview

In this section, we discuss the key design components of LeTS. The detailed design flow is shown in Algorithm 4.

▪ **Motivation.** In a real-time multi-task evaluation, an input query is evaluated by many fine-tuned transformers at the same time. Each one focuses on one specific sub-task and some tasks may depend on the computation result from others. For instance, multiple tasks exist in document editing software (e.g., Google Doc or Microsoft Word), such as analyzing tone, checking grammar, and then generating editing suggestions. Yet, the traditional fine-tuning method is extremely inefficient as the required computation and parameters grow linearly to the number of sub-tasks, which incurs the degraded quality of service and user experience. In this work, we aim to yield speedup through computation reuse for multi-task evaluation. Different from previous works [165, 8] that bypass computation in real-time, LeTS can generate a guaranteed speedup that is not input-dependent.

▪ **Limitation of traditional fine-tuning procedures.** We observe three limitations that hinge the parallelization and computation sharing in traditional fine-tuned procedures: (1) The computation of an attention layer can only start execution when all its previous layers yield the results. (2) Any modification of the bottom layers changes the subsequent computation, thus re-computation is required. (3) Although previous parameter-sharing work [53] can make W^{δ} sparse to reduce parameter growth in a sub-task, this sparsity cannot be exploited to reduce computation.

▪ **LeTS design.** Motivated by these observations, we propose a novel fine-tuning architecture that can reduce computation by reusing computed results. Also, the new architecture decouples the data dependency of different layers to enable speedup.

Given input query x_{in} (Figure 3.2(b)), LeTS first caches all N attention layers' output ($x_j^p, j \in \{0, 1, ..., N-1\}$) computed from input query x_{in} and pre-trained model W^p. For a given layer j in sub-task s, the input to the trainable layer W_j^f can be chosen from cached result x_{j-1}^p or the computed result x_{j-1}^f from the previous trainable layer. The attention output to the pooling

layer can be chosen from (i) x_j^p or (ii) x_j^f. LeTS uses pooling and Bi-LSTM to aggregate the outputs from attention layers to generate the final result. For each trainable layer W_j^f, we treat W_j^f as $W_j^p + W_j^\delta$ and make W_j^δ sparse using our proposed delta pruning algorithm.

We use an example architecture in Figure 3.2(b) to illustrate the advantages of the new architecture:

(i) **Bypass self-attention layers.** When the cached result x_j^p is used by the final pooling layer and the next trainable layer, the computation and parameters of the entire layer can be saved. This can be applied at layer W_j^p where $j \in \{0,1,2,6\}$.

(ii) **Exploit the sparsity of W^δ.** LeTS can leverage the sparsity feature of W_j^δ. More specifically, when the input to the trainable attention layer is x_{j-1}^p, LeTS computes $x_{j-1}^p \cdot W_j^\delta$ and adds it to a cached result. In Figure 3.2(b), when $j \in \{3,4,5,7,9\}$, the computation between x_{j-1}^p and W_j^{fK}, W_j^{fQ}, and W_j^{fV} (key/query/value parameters) can be reduced through exploiting this unstructured sparsity. Note that this sparse matrix multiplication can be easily implemented under many popular machine learning libraries [120, 150].

(iii) **Bypass linear layers in self-attention.** The pooling layer extracts the first hidden vector of each layer's output as the aggregate representation. For W_j^f where $j \in \{3,5,8,11\}$, only the first hidden vector of the output is used in the downstream computation; in this scenario, we move the pooling operation between n_3 and n_4 in Figure 3.2(a). As such, n_4 and n_5 in the self-attention layer can be reduced to a matrix-vector multiplication. The normalization layer (n_6) would be applied to only the pooling vector instead of the original layer output.

(iv) **Enable concurrent execution inside each transformer.** When the sub-tasks are dependent on each other and must be executed sequentially, the execution of our model can be paralleled across computing devices inside each transformer. This is because the parameter tuning on the bottom layers does not necessarily influence the downstream computation anymore. In Figure 3.2(b), assuming all nine GLUE tasks share the same architecture, the execution time is determined by the critical path (W_0^p-to-W_8^p+ $9\times W_9^f$-to-W_{11}^f), which will be $9T + 3T \times 9$ (3.0×

max speedup) compared to $12T \times 9$ of traditional $\text{BERT}_{\text{BASE}}$ fine-tuning (Figure 4(c)), assuming executing each attention layer takes time T.

A breakdown of the extra computation and parameter per task by leveraging (i)(ii)(iii) is shown in Figure 3.3. The computation and parameter overheads of the extra linear layers and Bi-LSTM are 0.01%/0.75% (Figure 3.3).

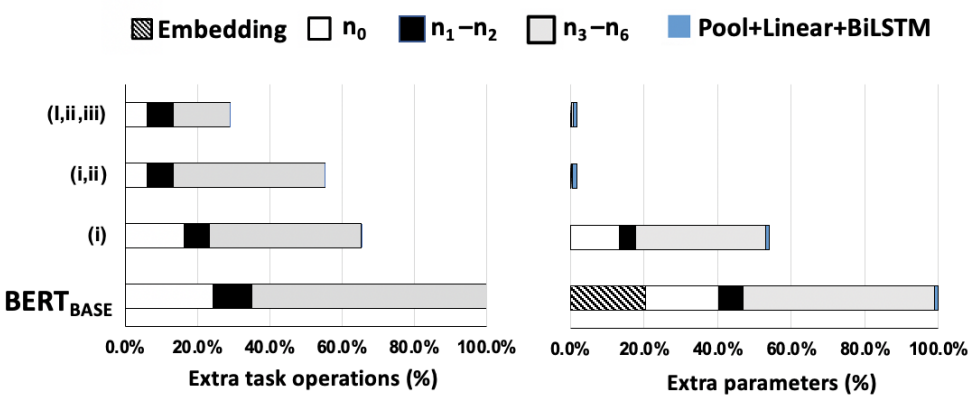


Figure 3.3. Extra computation and parameter breakdown leveraging (i) (ii) and (iii) for the example in Figure 3.2(b).

▪ **Neural architecture search for computation sharing.** All possible fine-tuning architectures of LeTS can be formulated into a search space as shown in Figure 3.4. Two NAS selectors are used in each layer to search (i) the input of the next layer (ii) the output to the final pooling layers.

The obtained architecture should achieve competitive sub-task accuracy with low extra computation operations. To address the problem, we leverage a differentiable NAS algorithm with a computation-aware loss to reflect both the computation cost and accuracy of a sub-task. The detailed searching algorithm is given in Sec. 3.3.

▪ W^{δ} **pruning.** Recent parameter-sharing approaches either add a new module between attention layers [115, 63] or generate a weight mask simultaneously during fine-tuning using l_0 normalization [53]. Yet, many works [45] have shown that l_0 regularization output is inconsistent for large-scale tasks. Also, the training parameters (i.e., weight mask and parameters) double

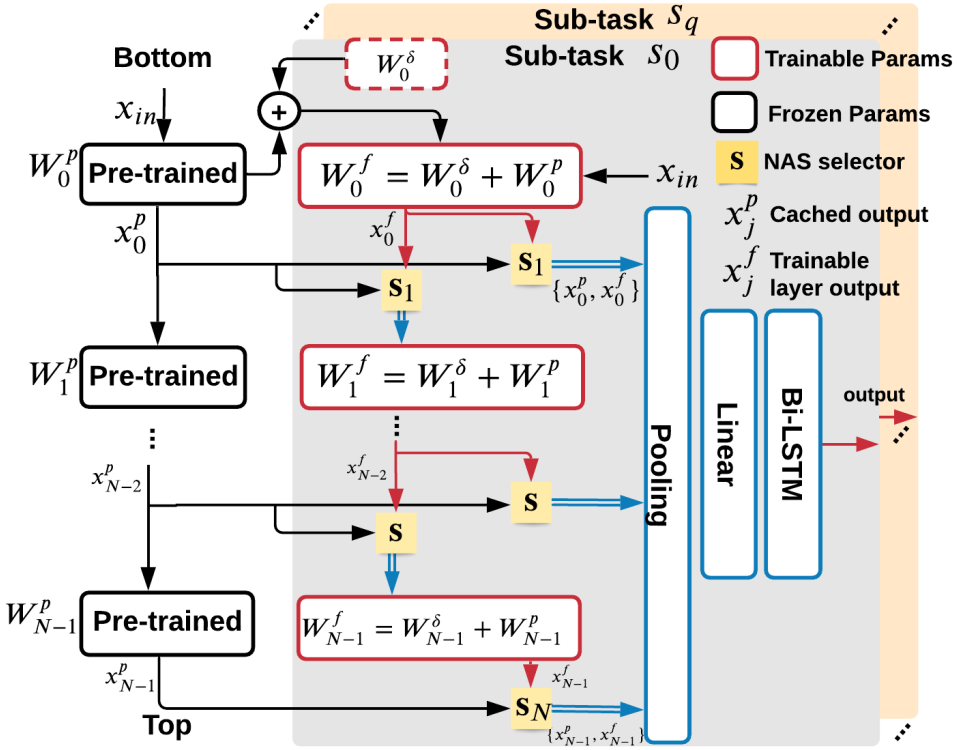


Figure 3.4. The search space of LeTS for computation and parameter sharing. x_j^p can be reused by all the sub-tasks.

during fine-tuning.

In this work, we treat the final fine-tuning weight W^f as the addition between pre-trained weight W^p and a weight difference (W^δ). By proposing an early-stage pruning approach, called *Delta-Pruning*, we compute the connection sensitivity of W^δ, which reveals the important connections in the W^δ for a given task (See Sec. 3.3.1). In this way, we can obtain the deterministic task-specific mask at the beginning of fine-tuning and use the generated mask to guide NAS.

Algorithm 3. LeTS Design Flow.

Input: : **Pre-trained model W^{p}; Preserving parameter number k; Group restriction G (Detailed in Sec. 3.4); Sub-task datasets $S = \{s_0, s_1, ..., s_q\}$.**

Output: : **Fine-tuning Policies $P_{\mathbf{out}}$ and Models $\mathscr{M}_{out}$.**

$\mathscr{M}_{out} \leftarrow \emptyset$, $P_{out} \leftarrow \emptyset$
for s_i in S **do**
  $W^{\tau} \leftarrow$ Generate_Search_Space(W^{p}, G)
  $c^{\tau} \leftarrow$ Delta_Pruning(W^{τ}, k, s_i) // c^{τ} is weight mask
  $M_i, P_i \leftarrow$ Computation_Aware_Searching$(W^{\tau}, c^{\tau}, s_i)$
  $c_i \leftarrow$ Delta_Pruning(W^{p}, M_i, k, s_i)
  $M_i \leftarrow$ Final_Finetuning(W^{p}, M_i, c_i, s_i)
  $M_{out} \leftarrow M_{out} \bigcup \{M_i\}$, $P_{out} \leftarrow P_{out} \bigcup \{P_i\}$
end for
return M_{out}, P_{out}

3.3 LeTS: Method

In this section, we detail the *Delta-Pruning* and Computation-aware neural architecture search algorithms in Algorithm 4.

3.3.1 Delta-Pruning in Early Stage

Delta-Pruning is motivated by SNIP [89] which targets to generate weight sparsity before training. We decompose the final fine-tuned weight (W^f) into two parts as shown in Eq. (3.1).

$$W^f = W^{\mathrm{p}} + \mathbf{c} \odot W^{\delta} \tag{3.1}$$

Here, $W^{\delta} \in R^d$ is the fine-tuning weight difference, $\mathbf{c} \in \{0,1\}^d$ is the generated mask for W^{δ}. $\odot$ is an element-wise product. Given a task dataset $\mathscr{D}$, the goal of Delta-Pruning is to find mask $\mathbf{c}$ at the beginning of fine-tuning without interfering with the searching and final fine-tuning phase. Assuming the k parameters in W^{δ} are preserved, the constrained optimization problem can be described as Eq. (3.2):

$$\begin{aligned} \min_{\mathbf{c},W^{\delta}} L(W^p + \mathbf{c} \odot W^{\delta}; \mathscr{D}) &= \min_{\mathbf{c},W^{\delta}} \frac{1}{n} \sum_{i=1}^{n} \ell(W^{\mathrm{p}} + \mathbf{c} \odot W^{\delta}; (x_i, y_i)) \\ s.t. W^{\delta} \in R^d, \mathbf{c} \in \{0,1\}^d,\ ||\mathbf{c}||_0 &\leq k \end{aligned} \tag{3.2}$$

Directly optimizing Eq. (3.2) using l_0 normalization will double the trainable parameters [103] and is unstable for large-scale tasks [45]. It is even more difficult to search l_0 masks together with architecture parameters in the DNAS algorithm (Sec 3.3.2). In this work, we intend to measure the effect of a connection e in W^{δ} on the loss function. Specifically, if removing W_e^{δ} does not show enough loss variation (ΔL_e), we set $c_e = 0$ to mask the gradient W^{δ} during training. Two challenges exist in computing ΔL_e: (i) Removing each connection in W_e^{δ} and checking the variation in loss is computation-consuming. (ii) W_e^{δ} is unknown at the beginning of fine-tuning. A random initialization method cannot reflect the fully fine-tuned W^{δ}.

To resolve (i), we relax the binary constrain on $\mathbf{c}$ to a continuous space and compute the

gradient of L with respect to c_e as g_e (Eq. (3.3)). Based on the intuition that the magnitude of derivative of c_e shows whether the parameter W_e^δ has a considerable effect on L or not, we use g_e to approximate ΔL_e when removing connection e in W^δ. As such, we define the connection sensitivity s_e for W_e^δ to be the g_e normalized by the sum of g_e in the network (Eq. (3.4)).

$$\Delta L_e(W^f;\mathscr{D}) \approx g_e(W^f;\mathscr{D}) = \left.\frac{\partial L(W_p + \mathbf{c}\odot W^\delta;\mathscr{D})}{\partial c_e}\right|_{\mathbf{c}=1} \tag{3.3}$$

$$s_e = \frac{|g_e(W^f;\mathscr{D})|}{\sum_{k=1}^{d}|g_k(W^f;\mathscr{D})|} \tag{3.4}$$

Then, assuming k parameters are pruned in W^δ, we generate mask $\mathbf{c}$ using a salient criterion computed from connection sensitivity s as Eq. (3.5):

$$c_e = 1[s_e - \widetilde{s_k} \geq 0],\ \forall e \in \{1...d\} \tag{3.5}$$

Here, $\widetilde{s_k}$ is the k-th largest element in the vector s and $1[\cdot]$ is the indicator function.

To resolve (ii), we first learn the weight difference initialization by warm-up the fine-tuning using $\mathscr{D}$ for steps N_{steps} and get $\widetilde{W}^f$. We then approximate W^δ using a task-specific initialization as $\widetilde{W}^\delta = \widetilde{W}^f - W^p$. Our ablation study shows that using task-specific warm-up shows better results compared to random initialization as this accumulation of gradients can better reflect the final weight difference distribution.

3.3.2 Differentiable Neural Architecture Search for Computation Sharing

As discussed in Sec. 3.2, a promising task-specific fine-tuning architecture should yield low extra computation costs and high task accuracy. We formulate the selection of fine-tuning model as a bi-level non-convex optimization problem as shown in Eq. (4.1).

$$\min_{a\in\mathscr{A}} \min_{w_a} \mathscr{L}(a, w_a) \tag{3.6}$$

Here, $\mathscr{A}$ is a new search space proposed in LeTS, $a \in \mathscr{A}$ is a set of continuous variables in the NAS selectors that specifies a possible architecture, W_a is the selected fine-tuning architectures from the search space $\mathscr{A}$ given a. The loss function $\mathscr{L}$ penalizes both accuracy degradation as well as the increase of extra computations.

▪ **Search space.** As shown in Figure 3.4, we decouple data dependency across layers by using a pooling layer, a linear layer, and a Bi-LSTM to aggregate all layers' output for final classification. The pooling layer uses the first hidden vector corresponding to the first token (i.e., [CLS] token) [38] as the layer presentation. The pooling output vectors are then fed into a linear layer and a Bi-LSTM.

LeTS first builds up a stochastic super network W^{τ} for the searching phase. Before searching, we copy the weights from W^{p} to W^{τ} trainable layers and disable the gradient computation based on c^{τ} which is the weight difference mask obtained from the Delta-Pruning. Two decisions should be made in each attention layer W_j^{τ}: (i) the input to the trainable layer. It can be either the cached result (x_{j-1}^{p}) or the output from the previous trainable layer (x_{j-1}^{f}) (ii) the output to the pooling layer from layer j. It can be either x_j^{p} or x_j^{f}. The total size of the search space would be 4^N where N is the layer number in the pre-training mode ($\approx 10^{15}$ for $\text{BERT}_{\text{LARGE}}$).

Two architecture selectors (s_{ij}, $i \in \{0,1\}$) are used to decide (i) and (ii) in layer j respectively ($j \in 1...N$ as shown in Figure 3.4). Each s_{ij} is associated with an architecture vector $\mathbf{a}_{ij}$ (1-by-2). We relax the choice of the architecture selection to a Gumbel Softmax [71] over the two possible sources:

$$\overline{x_j^{in}} = [x_{j-1}^{p}; x_{j-1}^{f}] \cdot Gumbel(\mathbf{a}_{ij}) \tag{3.7}$$

Here, $\overline{x_j^{in}}$ is the input to the jth trainable layer in W^{τ}. $[;]$ is a concatenation operation. *Gumbel* converts $\mathbf{a}_{ij}$ into a probability vector which is used to approximate discrete categorical selection. A temperature parameter T is associated with the *Gumbel* function to control its distribution. When T is high, $Gumbel(\mathbf{a}_{ij})$ becomes a continuous random variable and when T is low, $Gumbel(\mathbf{a}_{ij})$ is close to a discrete selection. During the search, we gradually lower T in *Gumbel*

to guide NAS.

▪ **Search algorithm and final fine-tuning architecture.** We alternatively update the two variables, i.e., $\mathbf{a}$ and W_a^τ under mask c^τ, to solve the bi-level optimization problem in Eq. (4.1). More specifically, we leverage second-order approximation [94] to update $\mathbf{a}$ since: (i) The total parameters in $\mathbf{a}$ is not large ($\sim$100). As such, it is feasible to use the second-order approximation although it requires more computation; (ii) second-order approximation can yield better solutions in NAS compared to gradient descent as shown in [94].

When searching ends, we choose the final connectivity using $\mathbf{a}$ in jth layer. Precisely, the input connectivity is chosen as $x_{j-1}^t = \arg\max_{t\in\{p,f\}} a_{0jt}$ and the output to the pooling layer is $x_{j-1}^t = \arg\max_{t\in\{p,f\}} a_{1jt}$. Then, we apply an optimization on attention layers where s_{1j} chooses x_{j-1}^f and $s_{0,j+1}$ chooses x_j^p to reduce even more computation. We move the final pooling operation between n_3 and n_4 in Figure 3.2(a). In this way, the computation of the following linear layers (n_4, n_5) can be reduced to a matrix-vector multiplication. The normalization layer (n_6) will be applied to the output vector from n_5.

▪ **Computation-aware loss function.** To consider both the task accuracy and computation cost, we define the loss function of LeTS's online searching phase as follows:

$$\mathscr{L} = CE(\mathbf{a}, W^\tau) \cdot \alpha log(E_{ops}(\mathbf{a}, W^\tau))^\beta \tag{3.8}$$

$CE(\mathbf{a}, W^\tau)$ is the cross-entropy loss given the architecture parameter $\mathbf{a}$ and the super net W^τ. α, β is the exponential factor that controls the magnitude of the operation terms. For the E_{ops}, we compute the expectation of operation over the architecture parameters $\mathbf{a}$:

$$E_{ops} = \sum\sum_j [Gumbel(\mathbf{a}_{0j}) \cdot Gumbel(\mathbf{a}_{1j})^T] \odot ops(W_j^\tau) \tag{3.9}$$

$ops(W_j^\tau)$ returns the number operations in layer j based on the selected combination of s_{0j} and s_{1j} into a 2-by-2 matrix. More specifically (See Figure 3.2(b)), (i) if both s_{1j} and s_{0j} select x_{j-1}^p as their input. then the computation of the entire layer j can be bypassed. (ii) If s_{0j} selects x_{j-1}^p

and s_{1j} selects x_{j-1}^{f} as its input, then the computation between x_{j-1}^{p} and W_j^{fK}, W_j^{fK}, W_j^{fK} would become sparse-matrix multiplication by leveraging the sparse feature of W_j^{δ}. The number of operations is computed according to the sparsity ratio of W_j^{τ}. Beyond the above two cases, the computation operations of a traditional self-attention layer are returned. Note that the *ops* returns a matrix of constant values given W_j^{τ}. As such, the E_{ops} is differentiable to the architecture parameters $\mathbf{a}_{0j}$ and $\mathbf{a}_{1j}$ to search for computation-efficient models.

3.4 The Evaluation of LeTS

3.4.1 Experiment setup

▪ **Datasets.** We evaluate LeTS on General Language Understanding Evaluation (GLUE) benchmark [157] consists of the following nine tasks: The Corpus of Linguistic Acceptability (CoLA). The Stanford Sentiment Treebank (SST-2). The Microsoft Research Paraphrase Corpus (MRPC). The Quora Question Pairs (QQP). The Semantic Textual Similarity Benchmark (STS-B). The Multi-Genre Natural Language Inference Corpus (MNLI) (We test on both matched domain MNLI_m and mismatched domain MNLI_{mm}). The Stanford Question Answering Dataset (QNLI). The Recognizing Textual Entailment (RTE).

▪ **Metrics.** We report Matthew's correlation for CoLA, Spearman for STS-B, F1 score for MRPC/QQP, and accuracy for MNLI/QNLI/SST-2/RTE, respectively. For computation efficiency, we report *max speedup* assuming the sub-tasks are dependent on each other. Also, we show *new FLOPs per task* over the FLOPs of a fine-tuned BERT and *total operations* that are required to compute the nine sub-tasks. For parameter efficiency, we report *total parameters* and *new parameters per task*.

Note that all previous parameter-sharing works cannot reduce the computation overhead for multi-task evaluation. Thus, we build two extra baselines: (vi) We freeze the bottom k self-attention layers and fine-tune the top layers. (vii) We append k new layers at the top of the pre-trained model and freeze the pre-trained weights (Figure 3.2(d)).

▪ **Detailed Explanation of Each Metrics.**

1) For *New operations per task*, we first compute the new operations (FLOPs or Floating-point operations) introduced by the sub-task based on the searched result and weight mask. We normalize the new operations of the task s_i (where $i \in \{1,...,N\}$) using the total number of operations required for a single $\text{BERT}_{\text{LARGE}/\text{BASE}}$ as Eq. (3.10):

$$ops_{\%}(s_i) = \frac{ops(s_i)}{ops(\text{BERT}_{\text{LARGE}})} \times 100\% \tag{3.10}$$

This percentage $ops_{\%}$ indicates that you only need extra $ops_{\%}$ new operations compared to the operations of an entire transformer (100%) when adding the sub-task to your multi-task system.

$$new\_ops\_per\_task(\%) = \frac{\sum_i^N ops_{\%}(s_i)}{N} \tag{3.11}$$

2) *Total operations (%)* include the extra operations from the nine GLUE tasks and the overhead operations from computing pretrained weight and input:

$$Total\_ops(\%) = \frac{\sum_i^N ops(s_i) + ops(overhead)}{ops(\text{BERT})} \times 100\% \tag{3.12}$$

For example, the total operations of the traditional fine-tuning method will be 9× (nine GLUE tasks) of the operation of $\text{BERT}_{\text{LARGE}}$ (100%×9=900%) and the ratio between *ops(overhead)* and ops($\text{BERT}_{\text{LARGE}}$) is 0%. For freezing bottom-12 layers, the new operations per task are 50% and the ratio between *ops(overhead)* and ops($\text{BERT}_{\text{LARGE}}$) is 50%. As such, the total normalized operations would be 40%×9 + 50% = 500%.

3) When the sub-tasks are independent of each other, the user can leverage the computation reduction to achieve speedup. Yet, in many cases, the sub-tasks are dependent on each other and must be executed in order. In this scenario, LeTS's design space can yield fruitful speedup as we decouple the computation of different attention layers inside each transformer. We first identify the critical path [59] of evaluating the 9 GLUE tasks. The *Max Speedup*, in this case, would be computed as:

$$max\_speedup = \frac{ops(BERT) \times N}{ops(critical\_path)} \tag{3.13}$$

Taking freezing Bot-12 as an example, ops(s_i)/ops($\text{BERT}_{\text{LARGE}}$) = 50%. Thus, the critical path would be the sum of overhead (50%) and the computation time (50%) for each sub-task (max_speedup = 900%/500% = 1.8×).

Note that both the freezing Bot-12 and original fine-tuning architecture are included

in our search space. Yet, the fine-tuning approach is computationally inefficient, and freezing Bot-12 sacrifices the task performance a lot. LeTS pushes the Pareto frontier between task performance and computation reduction when multiple tasks co-exist.

▪ **Baselines.** All previous parameter-sharing works are tested on $\text{BERT}_{\text{LARGE}}$ model [38]. We compare LeTS with the following baselines: (i) **Full fine-tuning** on $\text{BERT}_{\text{LARGE}}$ in a traditional way; (iii) **Adapter** [63]. (iv) **DiffPruning** [53]. (v) **BitFit** [125], which fine-tunes only the bias parameters using a large learning rate. (See Sec. 3.5)

Also, we compare LeTS with model compression works, such as **DistilBERT** , **MobileBERT** [145] and **TinyBERT** [72] which compressed the $\text{BERT}_{\text{BASE}}$ through knowledge distillation (Sec. 3.5). This comparison is conducted on $\text{BERT}_{\text{BASE}}$.

▪ **LeTS design settings.** We leverage LeTS to design fine-tuning models for platforms with different computing/storage budgets: (i) We search task-specific architectures for each task in GLUE and fine-tuning it with the generated sparse mask (denoted as **LeTS-(p,c)**[2]). (ii) During the final fine-tuning, we also conduct an ablation study by removing the weight mask to achieve better accuracy (denoted as **LeTS-(c)**). This is suitable for computing platforms with a low-cost storage budget. (ii) To maximize the parallelism in a searched model, we decouple the attention layers into **g** groups (denoted as **LeTS-G-g**) and require the first layer in each group to use the cached inputs (x^p_{j-1}); thus the evaluation of different groups can be executed concurrently. Inside each group, we still apply DNAS to decide the connections.

▪ **Hyperparameters.** The DNAS method takes 1 day on 4 V100 GPUs per task on average which is less than 0.5% of the pre-training cost of $\text{BERT}_{\text{LARGE}}$ [38]. The *max input length* for BERT is set to 128 to match previous baselines. Our pre-trained models and code base are from [163]. We use $N_{steps} = 100$ to initialize W^{δ} (Sec. 3.3.1). Inspired by [125] that the bias terms requires a larger learning rate to achieve better fine-tuning results, we apply two optimizers with different learning rate scheduler to update the bias terms ($lr_b \in \{1e^{-3}, 5e^{-4}\}$) and other parts ($lr_w \in \{2e^{-5}, 1e^{-5}\}$) separately during the final fine-tuning.

[2] **p** and **c** stand for parameter/computation-sharing

3.4.2 Results

▪ **Comparison to baselines on GLUE dataset.** Our comparison with the baseline methods is shown in Table 3.1. LeTS-(c)/-(p,c) can achieve similar performance (+0.2%/-0.2% on average) to a fully fine-tuned $\text{BERT}_{\text{LARGE}}$ model while saving 40.2%/49.5% computation. With a more aggressive setting, LeTS-G-4 can reduce 57.0% computation (3.84× speedup) while matching the task performance of Adapters. In the meantime, LeTS-(p,c)/-G-4 only adds 1.4% parameters per task (including the Linear and Bi-LSTM layers), which is more parameter-efficient than Adapters. LeTS illustrates a trade-off between concurrent execution speedup and multi-task performance which is not done by previous works.

Compared to DistilBERT_6, MobileBERT, and TinyBERT_6 that reduce the total computation by 50.4%/28.3%/50.4% on $\text{BERT}_{\text{BASE}}$, LeTS-G-3/-G-4 shows 56%/62% computation reduction while preserving a high task performance (-0.5%/-0.8%) compared to the fine-tuned model. For fiercely compressed models (e.g., TinyBERT-4, $\text{MobileBERT}_{\text{TINY}}$, DistilBERT-4), they show large performance degradation (-2.3%/-2.6%/-7.7%) compared to the full fine-tuning model although saving more computations than LeTS. Also, LeTS shows the lowest parameter overhead (1.15×) compared to all compression models.

Compared to 'Freezing Bot-12' and 'Appending Top-1', LeTS also achieves better task performance (+1.3%/+8.3%) on average. That is because we relax the fine-tuning constraints on all the layers and aggregate the results for the final classification. Also, when the number of sub-tasks increases, LeTS can save even more computations compared to Freezing Bot-12 (42.6% new FLOPs per task compared to 50%).

▪ **Comparison of Delta-Pruning with other parameter-sharing methods.** Table 3.2 shows the performance of *Delta-Pruning* compared to previous parameter-sharing works. To make a fair comparison, the result is tested using the original fine-tuning architecture. With the sparsity ratios restriction as 0.1%/0.25%/0.5% per task, LeTS achieves 2.4%/0.8/0.6% average performance increase compared to DiffPruning. This shows Delta-Pruning is effective in preserving task

accuracy compared to the l_0 regularization method.

3.4.3 Sensitivity and Ablation study.

▪ **Varying the sparsity constraint on $\text{BERT}_{\text{LARGE}}$ model / Weight mask distribution.** We also conduct a sensitivity analysis using Delta-Pruning with various sparsity ratios (0.1%/0.25%/0.5%) across GLUE benchmarks (Table 3.2). Different tasks show different sensitivity with the growth of the sparsity ratio. A better trade-off between accuracy vs. sparsity ratio can be achieved through grid search for each given task. Also, we show the distribution of weight masks for each layer varies across benchmarks (Figure 3.6). We hypothesize that when the tasks' inputs or outputs are related (e.g., QQP and QNLI both encode questions / MPRC and STS-B both generate similarity), they reveal similar mask distribution. This indicates that adding a uniform module (e.g., Adapter) between each layer for a task is sub-optimal.

▪ **Sensitivity to computation sharing ratio / Ablation to computation-aware loss function.** To show the capability of LeTS in reducing computation while maintaining a high task accuracy, we perform NAS for 2 tasks multiple times (with different α, β in the loss function) and sample different architectures from the distribution. Figure 3.5 shows the results between extra operations v.s. task accuracy (on GLUE dev set). Freezing bot-k layers cannot preserve task accuracy with the increase of k, while the architecture searched from LeTS does not show a large performance drop. LeTS also presents better task accuracy compared to the randomly sampled architectures, which shows that our DNAS algorithm can improve the quality of the searched model. When removing the computation-aware loss function, DNAS tends to select more trainable matrices to preserve task performance and the searched model cannot fully exploit computation sharing.

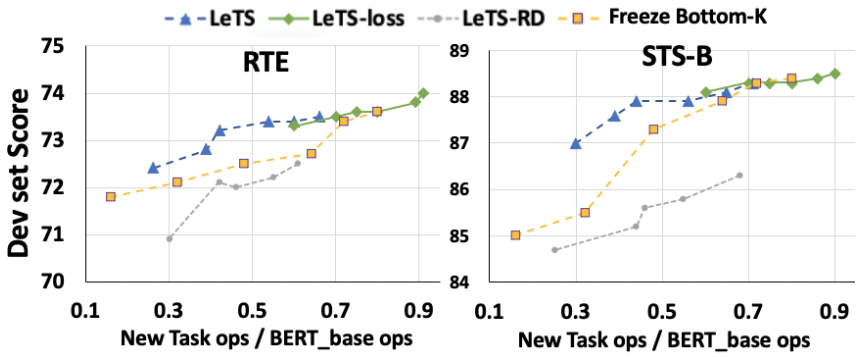


Figure 3.5. Sensitivity to computation sharing ratio (performed on $\text{BERT}_{\text{BASE}}$).

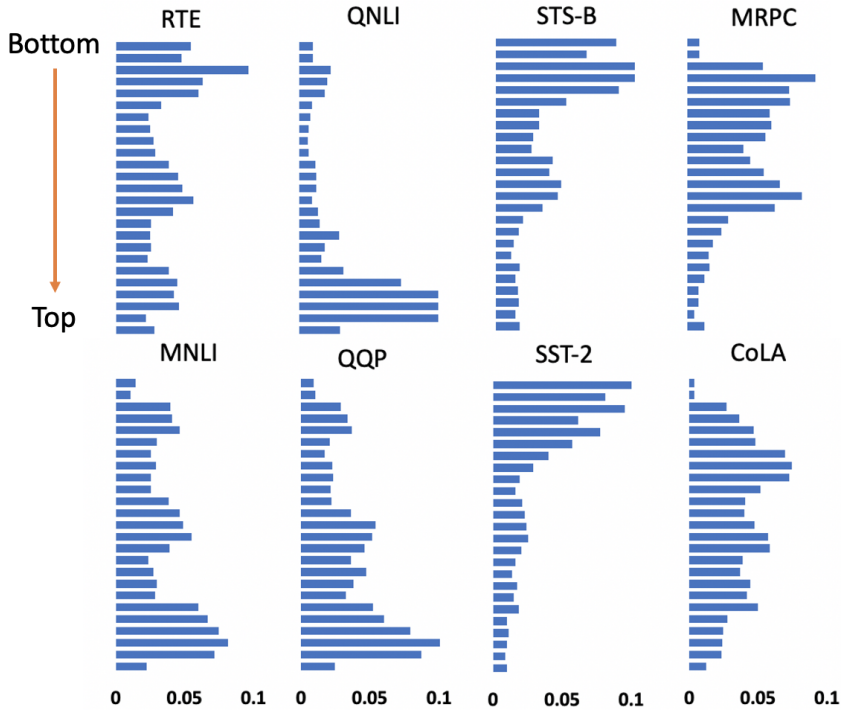


Figure 3.6. Distribution of LeTS's task-specific weight masks. (performed on $\text{BERT}_{\text{LARGE}}$)

Table 3.1. Comparison between LeTS and the baseline parameter-sharing works on $\text{BERT}_{\text{LRAGE}}$ using GLUE test set.

	Max Speedup †	New FLOPs Per Task% (Total FLOPs %)	Total Params %	New params Per Task	QNLI‡	SST-2	$\text{MNLI}_{\text{m}}/_{\text{mm}}$	CoLA	MRPC	STS-B	RTE	QQP	Avg
Full fine-tuning	1.00×	100% (900%)	9.00×	100%	92.7	**94.9**	86.7/85.9	60.5	89.3	86.5	70.1	**72.1**	**80.9**
Full fine-tuning *	1.00×	100% (900%)	9.00×	100%	**93.4**	94.1	**86.7/86.0**	59.6	88.9	86.6	71.2	71.7	80.6
Adapters (8-256)	1.00×	100% (900%)	1.32×	3.6%	90.7	94.0	84.9/85.1	59.5	89.5	86.9	**71.5**	71.8	80.4
Adapters (64)	1.00×	100% (900%)	1.18×	2.1%	91.4	94.2	85.3/84.6	56.9	89.6	**87.3**	68.6	71.8	79.8
Diff pruning	1.00×	100% (900%)	**1.05**×	**0.5%**	92.9	93.8	85.7/85.6	60.5	87.0	83.5	68.1	70.6	79.4
Diff pruning (struct.)	1.00×	100% (900%)	**1.05**×	**0.5%**	93.3	94.1	86.4/86.0	**61.1**	**89.7**	86.0	70.6	71.1	80.6
Freeze Bot-12	1.80×	50.0% (500%)	5.05×	45%	91.5	94.0	85.6/84.5	56.2	88.3	83.5	69.3	70.8	79.1
Freeze Bot-23	**6.75**×	**4.2% (138%)**	1.34×	3.7%	79.8	91.6	71.4/72.9	40.2	80.1	67.3	58.6	63.3	68.2
Append Top-1	**6.75**×	**4.2% (138%)**	1.34×	3.7%	82.1	91.9	75.7/74.6	43.4	83.4	81.2	59.8	66.6	72.1
LeTS-G-4 (p,c)	**3.84**×	**34.7% (387.3%)**	**1.13** ×	**1.4%**	92.5	93.8	85.3/84.8	59.8	88.6	86.4	70.8	71.1	80.1
LeTS (c)	2.60×	51.9% (537.9%)	6.66×	62.9%	**92.9**	**94.5**	**86.4/86.0**	**60.8**	**89.0**	**86.8**	**71.6**	**71.4**	**80.8**
LeTS (p,c)	2.84×	42.6% (454.2%)	**1.13**×	**1.4%**	92.6	94.2	85.5/85.1	60.4	88.9	86.5	71.4	71.1	80.4

* Fine-tuning results of our pre-trained $\text{BERT}_{\text{LARGE}}$ from huggingface [163].

‡ Besides DiffPruning and our results, previous works are reported on the old QNLI test set in the GLUE benchmark. The average is calculated without QNLI.

† When the sub-tasks are dependent on each other, max speedup can be achieved through concurrent execution.

Table 3.2. Sensitivity study to sparsity ratio constraint and comparison to parameter-sharing baselines on GLUE dev dataset.

	Total Params %	New params Per Task	QNLI	SST-2	$\text{MNLI}_{\text{m}/\text{mm}}$	CoLA	MRPC	STS-B	RTE	QQP	Avg
Full fine-tuning	9.00×	100%	**93.5**	94.1	**86.5/87.1**	62.8	**91.9**	**89.8**	71.8	**87.6**	**85.0**
Diff-Pruning (struct.)	1.01×	0.1%	92.7	93.3	85.6/85.9	58.0	87.4	86.3	68.6	85.2	82.5
Diff pruning (struct.)	1.03×	0.25%	93.2	**94.2**	86.2/86.5	63.3	90.9	88.4	71.5	86.1	84.5
Diff pruning (struct.)	1.05×	0.5%	93.4	**94.2**	86.4/86.9	**63.5**	91.3	89.5	71.5	86.6	84.8
BitFit*	1.01×	0.08%	91.1	93.3	-	62.9	91.5	89.5	**75.1**	**87.6**	-
LeTS (p)	1.01×	0.1%	92.3	93.3	85.3/85.7	63.5	91.6	89.6	75.1	87.4	84.9
LeTS (p)	1.03×	0.25%	92.7	93.9	85.9/86.2	64.1	91.7	**89.8**	75.7	**87.6**	85.3
LeTS (p)	1.05×	0.5%	**92.9**	**94.0**	**86.4/86.2**	**64.4**	**91.9**	**89.8**	**75.8**	**87.6**	**85.4**

* BitFit does not report all the performance on GLUE dev set.

Table 3.3. Comparison between LeTS and the model compression works on $\text{BERT}_{\text{BASE}}$ using GLUE test set.

	Max Speedup	New FLOPs Per Task% (Total FLOPs %)	Total Params over $\text{BERT}_{\text{BASE}}$	Total Params	QNLI*	SST-2	$\text{MNLI}_{\text{m}}/_{\text{mm}}$	CoLA	MRPC	STS-B	RTE	QQP	Avg
Full-finetuning	1.00×	100% (900%)	9.00×	110M × 9	90.9	93.4	83.9/83.4	52.8	87.5	85.2	67.0	71.1	78.0
DistilBERT_4	3.00×	33.5% (300%)	4.27×	52.2M × 9	85.2	91.4	78.9/78.0	32.8	82.4	76.1	54.1	68.5	70.3
$\text{MobileBERT}_{\text{TINY}}$	8.55×	13.6% (123%)	1.24×	15.1M × 9	89.5	91.7	81.5/81.6	46.7	87.9	80.1	65.1	68.9	75.4
TinyBERT_4	9.40×	5.2% (46.8%)	1.19×	14.5M × 9	87.7	92.6	82.5/81.8	44.1	86.4	80.4	66.6	71.3	75.7
DistilBERT_6	2.00×	49.5% (446%)	5.48×	67.0M × 9	88.9	92.5	82.6/81.3	49.0	86.9	81.3	58.4	70.1	75.3
MobileBERT(-OPT)	1.78×	71.7% (645%)	2.07×	25.3M × 9	91.6	92.6	84.3/83.4	51.1	88.8	84.8	70.4	70.5	78.2
TinyBERT_6	2.00×	49.5% (446%)	5.48×	67.0M × 9	90.4	93.1	84.6/83.2	51.1	87.3	83.7	70.0	71.6	78.1
LeTS-G-3 (p,c)	2.83×	36.7% (397%)	1.15×	127M	90.4	92.2	82.8/81.8	50.1	88.3	84.6	70.0	70.1	77.5
LeTS-G-4 (p,c)	3.77×	27.6% (323%)	1.15×	127M	90.0	92.0	82.6/81.6	49.9	87.9	84.5	69.5	69.8	77.2

* The average is calculated without QNLI.

3.5 Related Work

Fine-tuning for transfer learning. Transferring a pre-trained model to a task of interest can be achieved by fine-tuning all the weights on that single task [64]. Recent advances in text classification [33, 98, 73, 168] have been achieved by fine-tuning a pre-trained transformer [155]. However, it modifies all the weights of the network which is parameter inefficient for downstream tasks.

Multi-task learning. Multi-task learning (MTL) learns models on multiple tasks simultaneously and utilizes them across a diverse range of tasks [19]. MTL has been widely exploited using BERT and shows good performance on multiple text classification tasks [96, 28]. In this work, we assume multiple tasks arrive in stream (i.e., online setting) and thus jointly training is not available as discussed in Sec. 3.1. Moreover, it is challenging to balance multiple tasks and solve them equally well in training [143]. LeTS can be potentially combined with MTL. This will change our assumption of online settings (task arrives one-by-one) to 'n tasks arrive at the same time'. The n tasks will be searched and fine-tuned together. We leave this combination as a future work.

Parameter sharing for fine-tuning. Adapter is an alternative for parameter-efficient BERT models for online settings [63]. It works well on machine translation [9], cross-lingual transfer [154], and task composition for transfer learning [115]. These task-specific adapters are inserted between layers and cannot exploit the computation sharing because of the modification on the bottom layers. Recent work [53] use l_0 normalization to train a mask during fine-tuning for multi-task NLP. In this section, we propose a novel method to prune weight difference based on SNIP [89] to condense the task-specific knowledge which achieves better parameter efficiency.

Hardware-aware NAS. Recent advances in NAS leverage differentiable methods by relaxing the selection of architectures in a continuous space to reduce the high search cost of RL-based NAS [185, 147, 148]. Previous differentiable NAS work [164, 94, 156, 43] mainly focus on computer vision tasks. In LeTS, we combine the searching algorithm of [164] and

[94] to resolve a unique problem in NLP. LeTS also presents a novel search space with a computation-aware loss function to search model with high task accuracy and computation sharing ratio.

Model compression. Model pruning is another way to reduce DNN model size and computation. [51] prune weights in BERT based on magnitude, [54] use iteratively reweighted l_1 minimization, and [85] leverage cross-layer parameter sharing. Other works distill knowledge from a pre-trained model down to a smaller student model [132, 145, 72]. Note that LeTS is orthogonal to all these methods, as we did not modify the pre-trained parameters. we leave this combination as future work.

3.6 Conclusion

We propose LeTS, a transfer learning framework that achieves computation and parameter sharing when multiple sub-tasks co-exist. LeTS proposes a novel architecture space that can reuse computed results to reduce computation. By leveraging NAS with a computation-aware loss function, LeTS can find models with high task performance and low computation overhead. By treating the fine-tuned weight as the sum of pre-trained weight and weight difference, we present an early-stage pruning algorithm to compress weight difference without task performance decrease. The integration of the above novelty enables even more computation reduction by exploiting the sparsity of the weight difference. LeTS achieves better task performance compared to previous parameter-sharing only methods. Also, by leveraging computation sharing, LeTS engenders large computation reduction to enable scalable transfer learning. LeTS can be combined with multi-task learning to achieve better task performance and we leave it as a future work.

Acknowledgement. Chapter 3, in part, contains a re-organized reprint of the material as it appears in International Conference on Machine Learning (ICML) 2021. Fu, Cheng; Hanxian Huang; Xinyun Chen; Yuandong Tian; Jishen Zhao. The dissertation author was the primary

investigator and author of this paper.

Chapter 4

Designing DNNs with Model Parallelism for Multi-device System

4.1 Inefficiency of Existing Neural Architecture Search Approach

Deep neural networks (DNNs) are increasingly adopted in various fields due to their unprecedented performance. Enormous architecture-level advancements have been proposed to improve the performance and efficiency of DNN execution [146]. In the meantime, Neural architecture search (NAS) [186, 185] enables a new line of research that aims to design efficient DNN topology for target hardware platforms. Existing hardware-friendly NAS techniques [164, 17, 147] target to identify efficient models on single CPU/GPU systems by introducing the notion of *FLOPs*, model size, or predicted latency into the searching process. However, in modern real-time and cloud computing scenarios, multiple computation components co-exist and are shared by different tasks or users.

There are two types of parallelism when executing a NN on a multiple-device platform. (i) Model parallelism partitions operations into different parts and assigns them to different devices. (ii) Data parallelism distributes the inference data onto multiple devices and duplicates the entire model on each device. However, data parallelism has several limitations: (i) Data parallelism can increase the throughput of the application but not the end-to-end inference latency, thus it is not suitable for real-time applications. (ii) Data parallelism may incur severe communication

overhead during the setup stage since the inputs need to be distributed to each available device. (iii) Data parallelism is infeasible when the model size is too large to fit on a single device with constrained memory. With the increasing model size of SOTA neural networks, the fact in (iii) is becoming more stringent, especially for edge devices with very limited memory. In contrast, emerging multi-device computing systems and device-to-device communication techniques (e.g, 5G communication, NVLINKs, and PCIe Gen 4) can benefit model parallelism. Because our goal is to reduce application latency, model-level parallelism is a promising candidate.

The designed search spaces of conventional NAS can be classified into two main categories (see Figure 4.2): (i) **Layer-wised search.** Existing latency-oriented model searching methods [164, 17, 17] explore model architectures that are analogous to *chain-like* wired mobilenetV2 with the building block shown in Figure 4.2(a). This topology is hard to parallel across multiple computation components due to data dependency. (ii) **Cell-structure search.** Another line of methods [186, 94] proposes to search a building block, called *cell*, on small tasks (e.g., CIFAR-10) and transfer the searched cell for training on a large task (e.g., ImageNet). The model in this space has certain levels of parallelism due to the concurrent execution of different blocks inside each cell. However, the concatenation (*concat*) operations between cells prevent further parallelism since they synchronize the execution and increase communication overhead. To overcome these constraints, ColocNAS is motivated to automate the design of suitable neural network architectures for multi-device platforms to achieve low inference latency and high task accuracy.

Designing an efficient model to achieve model parallelism across devices is a non-trivial task. It is challenging since: (C1) The searched architecture needs to keep a high accuracy while reducing its runtime overhead. (C2) While neural networks with more complicated wiring patterns have a higher chance for model parallelism, predicting their execution time is challenging for latency-aware NAS. (C3) The actual latency of a specific network architecture in a multi-device setting depends on the device placement policy, which is unknown ahead of time.

To tackle these challenges, we propose *ColocNAS*, a differentiable NAS framework that

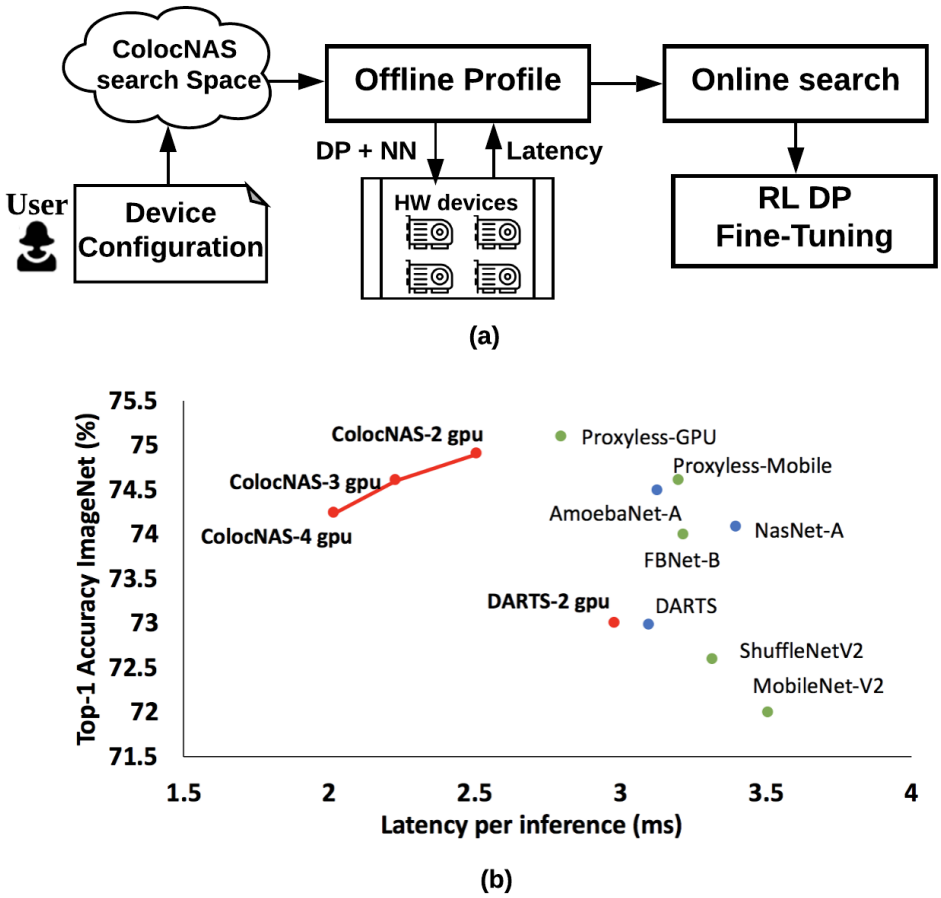


Figure 4.1. (a) ColocNAS Design overview and (b) roadmap between accuracy and latency of NAS methods. *DP+NN* refers to device placement (DP) policy and neural network (NN) architecture for deploying. ColocNAS significantly outperforms other NAS methods, when running on multiple devices.

effectively searches network architectures for the given hardware environment. ColocNAS resolves all the aforementioned challenges (C1-C3) using the following solutions (as shown in Figure 4.2 (a)): (S1) ColocNAS designs a new search space with less synchronization and communication overhead by exploring elaborate connectivity patterns in the NN. (S2) Based on the observation that similar computation graphs yield closer latency, ColocNAS leverages offline latency profiling and *k-Nearest Neighbors* (kNN) with graph similarity to predict latency for the searched model in the online phase. (S3) ColocNAS employs uniform workload distribution as the *expert placement policy* to facilitate offline latency profiling and to guide online latency-

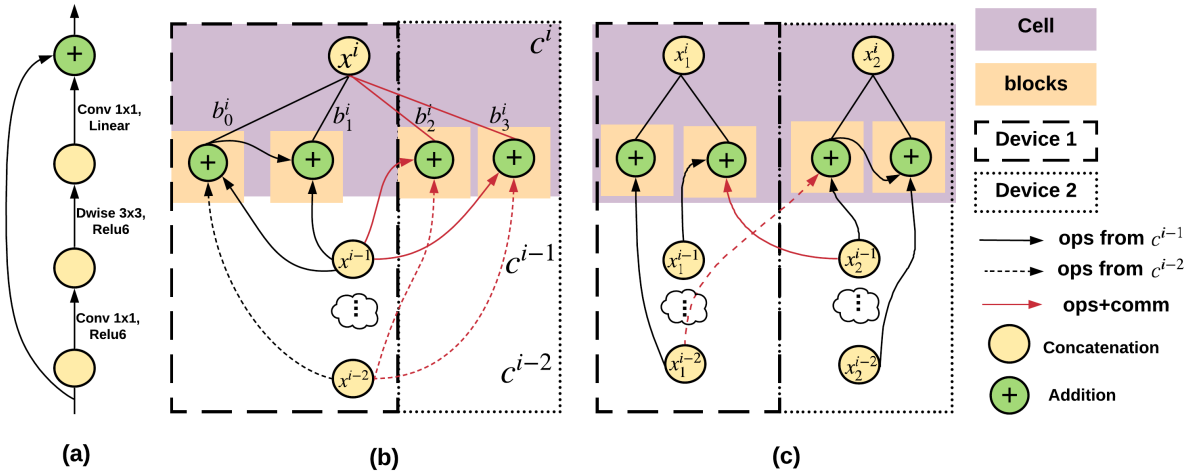


Figure 4.2. An example of basic building blocks in three types of search space. (a) layer-wise searched model (b) a cell-based searched model (c) a new search space proposed in ColocNAS. The yellow circle refers to the concatenation node. b_n^i denotes the n^{th} block in i^{th} cell (i.e., layer).

aware NAS. This policy leverages the intrinsic structure features of the DNN in the new search space. After online NAS, the placement of the searched model is further fine-tuned using a Reinforcement Learning (RL)-based algorithm to reduce its runtime on the target platform.

Our framework offers a holistic solution to designing efficient and accurate neural networks on multi-device systems. Extensive experiments show that ColocNAS reduces inference latency by a large margin while achieving competitive accuracy and latency as shown in Figure 4.1(b). Our work sheds light on a new dimension (device-level parallelism) of hardware-aware NAS algorithms.

4.2 Related Work

4.2.1 Neural Network Accelerator

There has been a line research that targets to develop specific hardware for efficient NN execution. The runtime and energy consumption bottleneck of existing NN accelerators are induced by memory access. As such, designers are seeking to shift expensive DRAM access into low-cost device-to-device communication. To meet this need, the new generation of V100 GPU has NVLinks that are hardware linkers for direct data communications between devices. TPUs are also equipped with high-speed network connection across devices [74]. These emerging new data communication technologies will greatly reduce the latency and energy overhead due to intensive DRAM access, thus making model parallelism more efficient.

4.2.2 Neural Architecture Search

Prior works [186] used RL methods for NAS. However, these methods are computationally expensive, as the rewards are obtained by sampling different architectures and training them to acquire accuracy. Recent works also try to reduce searching time using weight sharing mechanisms [10, 93] or gradient-based methods [94, 164]. ColocNAS combines two gradient-based methods [164, 94] by using an approximated second-order derivative to train the architecture parameters and converting them into probability using and *Gumbel Softmax* equation [164] for architecture selection. Previous cell-structured NAS works [94, 186] employ CIFAR-10 as a proxy for searching normal cells and reduction cells, then transfer the searched blocks to form a large network for CIFAR-10 and ImageNet evaluation. ColocNAS also applies this optimization mechanism to reduce searching time.

For hardware-aware NAS, existing works [164, 147, 17] target to optimize the latency on a single GPU/CPU device with a layer-wise search space. As such, the runtime latency of a given neural architecture is approximated using a pre-trained neural-based latency predictor [17], or by summing up the latency of each layer obtained from a pre-profiled look-up table [164]. As

our search space generates much more complex wired networks, we propose a new end-to-end latency estimation method that predicts the inference runtime after device placement.

4.2.3 Neural-based Device Placement

After the proper architecture is found by ColocNAS, we need to place the operations in the model onto the target hardware system. The expert placement policy (uniform partition) can be further optimized using RL algorithms [105]. ColocRL [105] leverages the policy gradient to optimize the placement latency.As none of this work is open-sourced, we rebuild the *ColocRL* (policy gradient) method for device placement to reduce the runtime latency.

4.3 ColocNAS Design

ColocNAS uses *differentiable neural architecture search* (DNAS) by combining two gradient-based methods [164, 94] to solve the problem of topology design. We formulate the neural architecture search problem as a non-convex optimization problem as shown in Eq. (4.1). A promising architecture yields small latency and high task accuracy:

$$\min_{\alpha \in A} \min_{w_\alpha} L(\alpha, w_\alpha) \tag{4.1}$$

Here, A is a new search space proposed in ColocNAS, $\alpha \in A$ is a set of continuous variables that specify a possible architecture, w_α is the weight parameter of the network. L is the loss function that penalizes both accuracy degradation as well as the increase of inference latency. The design flow of ColocNAS is detailed in Algorithm 4.

Algorithm 4. ColocNAS Design Flow.

Input: Prototyping Hardware Devices (τ)**; Device Number** $(\mathscr{N}_\tau)$**; ArchTable Size (**N_{arch}**);**
Cell Number N_c**; Possible Operations (**Ops**).**
Output: Fine-tuned Device Placement Policy P_{RL}**; Model** M_{out}**.**
Offline Profile:
1: **for** $i = 1, ..., N_{arch}$ **do**
2: $M_i \leftarrow Random\_Generate(\mathscr{N}_\tau, Ops, N_\tau, N_c)$
3: $P_i \leftarrow Expert\_Placement\_Policy(M_i)$
4: $Arch\_Table \leftarrow Hardware\_Profile(M_i, P_i, \tau)$
5: **end for**
Online Searching and Training:
6: $M_{arch} \leftarrow Searching(Arch\_Table, Ops, N_\tau, N_c)$
7: $M_{out} \leftarrow Training\_Model(M_{arch})$
RL Fine-Tuning:
8: $P_{RL} \leftarrow RL\_FineTuning(M_{out}, \tau)$

4.3.1 Search Space

Previous works search models with layer-wise or cell-based structures as shown in Figure 4.2 (a) and (b). The layer-wise structure is suitable for a *single* CPU device as the identified model topology is very regularized and its execution is fast due to data locality.

However, the resulting chain-like model is not suitable when multiple devices are available for parallelism, as each layer can only start execution when the outputs of all its previous layers are computed. For the cell structure search space, its evaluation can be paralleled. Yet, the *concatenation* at the end of each cell node works as a *synchronization* unit that collects all the results inside the cell blocks before starting the computation in the next cell. This *synchronization* incurs severe communication overhead (the red arrow in Figure 4.2) and hinders computation parallelism. As such, we proposed a new search space that reduces the participants of the synchronization unit in order to minimize the delay of computation as shown in Figure 4.2(c). ColocNAS uses the same definition of search blocks and cells while introducing a new block connectivity pattern by 'duplicating' the concatenation nodes.

The traditional cell consists of an ordered sequence of N blocks where each block has two input edges (i, j) as shown in Figure 4.2(b). An edge can be selected from two sources: (i) previous blocks in the current cell; (ii) the output of the previous two cells. Each edge is associated with an operation that is determined after the search ($o(i, j)$) is performed. Assuming the inputs of the block are x^1 and x^2, the output of the block is computed as follows:

$$b_j = \sum_{i=1}^{2} o^{(i,j)}(x^i) \tag{4.2}$$

The intermediate results from the N blocks are concatenated together to form a single cell output x^i. Since we identify that this concatenation node is the bottleneck for achieving device-level parallelism, the new search space is defined as follows.

Instead of reducing towards a single concatenation node in each cell, the blocks of the ith cell will be evenly connected to C different concatenation nodes, resulting in outputs $x^i = [x^i_1, x^i_2, ..., x^i_c]^T$. For the nth block in the ith cell, we define *block connectivity* parameters β^{i-1}_n as the probability over each possible connection between the current block to any concatenation node in $(i-1)^{th}$ layer. It will be used to compute the weighted sum input $\overline{x^{i-1}_n}$ from previous layer $i-1$ as:

$$\overline{x_n^{i-1}} = softmax(\beta_n^{i-1}) \cdot x^{i-1}$$
$$= \sum_{c=1}^{C} \frac{exp(\beta_{n,c}^{i-1})}{\sum_{c'=1}^{C} exp(\beta_{n,c'}^{i-1})} x_c^{i-1} \tag{4.3}$$

Here β_n^{i-1} is a vector of size 1-by-C. $\overline{x_n^{i-2}}$ is computed from β_n^{i-2} using the same method as Eq. (4.3).

Note that ColocNAS explores optimal network architectures by solving the bi-level optimization problem defined in Eq. (4.1). The optimization variable $\alpha = \{\beta, ops\}$ where β is the block connectivity parameter and the softmax of ops is the probability vectors over the pre-defined set of DL operations for every possible connection (The candidate operations are the same as [17]). At the end of the search, we choose the connectivity of the nth block in ith cell from the previous two cells as $x_c^{i-1} = \arg\max_c \beta_n^{i-1}$ and $x_c^{i-2} = \arg\max_c \beta_n^{i-2}$.

Through comparison experiments, we find that using Gumbel softmax [164] over β_c^i in Eq. (4.3) can improve both the accuracy and latency of the searched model compared to the softmax function.

Given $\overline{x_n^{i-1}}, \overline{x_n^{i-2}}$ as the computed input from the previous two cells for the ith cell, the output of each block b_i^n would be the weighted sum over ops [94] during architecture search.

4.3.2 Offline Hardware-bounded Latency Profiling

ColocNAS integrates a latency-aware, gradient-based NAS for the target devices into the searching process to make the low latency model preferable. Recall that the optimization variable α in our search space is a set of probabilistic variables, thus the inference latency given a specific choice of α is also a random variable lat. We use the *expectation* value of the latency variable to assess the quality of the architecture choice α. However, computing the latency expectation over a architecture probability $E_{lat}(p_\alpha)$ on the given devices is challenging since: (i) Measuring the real latency value of each architecture during the searching process is infeasible due to the prohibitive latency cost. (ii) The latency of a complex graph is hard to predict while considering

the delay incurred by data movement. (iii) The latency value of an architecture choice is highly dependent on the placement policy, which makes it hard to compare the optimal latency values of different network topologies. To resolve the above problems, the first stage of ColocNAS is an *offline latency profiling* step that measures the inference time of diverse network architectures on the target devices.

Besides, to disentangle the complexity of model placement on the given hardware, we use a pre-defined *expert placement policy* for all searched architectures. In particular, our expert placement policy uniformly distributes each concatenation node and the associated operations onto all existing devices. By doing so, we reduce the overhead of synchronization and the cross-device communication compared to the traditional cell-based search space as shown in Figure 4.2(c).

It is also intractable to profile the latency values of all the architectures in the entire search space ($\sim 10^{20}$ models). Leveraging the observation that similar neural networks yield closer runtime latency, we propose a method to *approximate* the latency of a complicated wired neural network using *k Nearest Neighbour (kNN)*. We randomly generate architectures and profile their latency values on the target hardware using *expert placement policy* to obtain a lookup table $ArchTable$. The distance between two neural networks can be measured by using *graph edit distance* [46]. To approximate the latency expectation over an architecture parameter, we first convert α into the corresponding probability variable p_α which includes the following two parts:

$$p_{ops} = Softmax(ops), \tag{4.4}$$

$$p_\beta = GumbelSoftmax(\beta). \tag{4.5}$$

Given a specific choice of the probability parameter $p_\alpha = \{p_\beta, p_{ops}\}$, we sample N neural network architectures $arch_1, arch_2, ..., arch_N$. The expectation value of the latency variable is then approximated as:

$$E_{lat}(p_\alpha) \approx \frac{1}{N}\sum_{n=1}^{N} kNN(arch_n, ArchTable). \tag{4.6}$$

where the $kNN(\text{cot})$ function returns the average latency of the top-k closest architecture latencies for input $arch_n$.

4.3.3 Online Latency-aware Architecture Searching

ColocNAS aims to find a network architecture with low latency and high task accuracy. As such, we define the loss function of ColocNAS's online searching phase as follows:

$$L(\alpha, w_a) = CE(\alpha, w_a) + \lambda_1 L_{lat}, \tag{4.7}$$

$$L_{lat} = exp(-||p_\alpha - \widehat{p_\alpha}||_2) \cdot E_{lat}(p_\alpha). \tag{4.8}$$

Here, $CE(\alpha, w_a)$ is the cross-entropy loss given the architecture parameter α and the weights w_a. λ_1 is the scaling factor within the range $[0,1]$ that controls the trade-off between accuracy and latency. $E_{lat}(p_\alpha)$ is the expected latency of the specific architecture parameter obtained from Eq. (4.6). For the latency term, we add a regularization term $exp(-||p_\alpha - \widehat{p_\alpha}||_2)$ where $\widehat{p_\alpha} = \frac{1}{N}\sum_{n=1}^{N} arch_n$ to make the latency term differentiable to the architecture parameters α. If the sample mean value of the architecture probability $\widehat{p_\alpha}$ is far from the true one p_α, the corresponding latency term will be weighted less. Note that after the model is searched, we train the model from scratch to obtain the final accuracy.

4.3.4 RL-based Device Placement Fine-tuning

Once the model is searched and trained, the model can be directly deployed onto the target hardware systems using the expert placement policy that uniformly distributes the workload to each available device. However, this heuristic-based device placement policy may not be optimal considering the heterogeneous property of the underlying computing platforms. To further optimize the latency of the searched architecture Γ, the last step of the ColocNAS leverages

policy gradient for device placement on the given hardware. We apply the RL-based method proposed in [105] where the policy network is an attentional auto-encoder that takes the neural network graph as input. Each input in the encoding sequence is the concatenated embedding of the operation type, connectivity, and output shape of each block in the graph. The decoder sequentially generates the placement policy P. The goal is to learn the policy $\pi(P|\Gamma;\theta)$ to minimize the objective $J(\theta) = E_{P\sim\pi(P|\Gamma;\theta)}[R(P)|\Gamma]$. Here, $R(P)$ is the expectation of the reward (execution time).

The policy gradients are computed via the REINFORCE equation [161] as follows:

$$\nabla_\theta J(\theta) = E_{P\sim\pi(P|\Gamma;\theta)}[R(P)\cdot\nabla_\theta log_p(P|\Gamma;\theta)] \tag{4.9}$$

By sampling K placements following the placement policy probability $P\sim\pi(\cdot|\Gamma;\theta)$, the expectation of reward $R(P)$ can be estimated and the policy gradient is computed as:

$$\nabla_\theta J(\theta) \approx \frac{1}{K}\sum_{i=1}^{K}(R(P_i)-B)\cdot\nabla_\theta log_p(P|\Gamma;\theta) \tag{4.10}$$

Here B is the mean value of the rewards computed from the K sampled placements. ColocNAS leverages the intrinsic structure of the architecture in the new search space and performs device placement only for blocks and concatenation nodes (Figure 4.2 (c)). As a result, the training of the policy network $\pi(P|\Gamma;\theta)$ converges much faster compared to the original one in [105].

4.4 ColocNAS Evaluation

In this section, we demonstrate ColocNAS's effectiveness of model-level parallelism and accurate classification performance on CIFAR-10 and ImageNet tasks compared to the state-of-the-art NAS methods.

4.4.1 Experimental Setup

We run ColocNAS on different device configurations and evaluate the searched architectures in terms of its test accuracy and inference latency.

Searching phase setup. ColocNAS searches two types of cells, namely, a *normal cell* and a *reduction cell*. We put the reduction cell at $1/3$ and $2/3$ location of the neural architectures. After each reduction cell, we double the number of output channels in the network. The operations searched in the reduction cell has stride$=2$.

The model that uses on the searching phase has 8 cells. Larger networks can be built by stacking multiple normal cells in between the reduction cells. We alternatively update the two variables (α and w_α to solve the bi-level optimization problem in Eq. (4.1). In particular, we use second-order approximations to update the architecture parameter α. All possible operations in the search space and relevant searching details are the same as DARTS [94].

Hardware platforms and corresponding search space. To prove the generality of ColocNAS on different platforms, we test our method on two types GPUs and 6 different device configurations: (i) 2/3/4 Tesla K80 GPUs with Gen3 PCIe device connection. (ii) 2/3/4 Tesla V100 GPUs with NVlink connections.The device placement algorithm and latency profiling for cell structure are implemented in Tensorflow v1.14. For each device setting, we set the number of concatenation nodes to be the same as the number of devices to facilitate the *expert placement policy*. Since the device configuration determines the search space as shown in Figure 4.2, we name the search space with 2/3/4 GPUs as *ColocNAS-SP-2*/*ColocNAS-SP-3*/*ColocNAS-SP-4*. We denote the searched architectures as *ColocNAS-b4c2*/*ColocNAS-b6c3*/*ColocNAS-b4c4* which have 4/6/4

number of blocks and 2/3/4 concatenation nodes in each cell for hardware settings with 2/3/4 GPUs, respectively.

4.4.2 Latency Prediction Results

Recall that ColocNAS applies the kNN method to estimate end-to-end latency. We sample 10,000 architectures that are mapped using the expert placement policy to available GPUs on both CIFAR-10 ($32\times32\times3$) and ImageNet ($224\times224\times3$) datasets. We set the total number of cells to be 14 and 20 for ImageNet and CIFAR-10, respectively. For CIFAR-10 and ImageNet for evaluation, we added one/two initial stem cells to downscale the raw images, respectively.

Figure 4.3 shows the effectiveness of our kNN-based latency predictor. The latency is profiled in search space *ColocNAS-SP-4* with ImageNet input (batch size is 32) on 4 Tesla K80 GPUs. We use 5% of the profiled data for testing. The profiling takes 2.5 days on 8 GPUs. The RMSE error is $0.22ms$ on the test data, which is adequate for the latency prediction of complicated graphs. Note that this off-line profiling is a one-time profiling process, that can be used to search different NN architectures when tuning the λ_1 parameters in the Eq. (4.7).

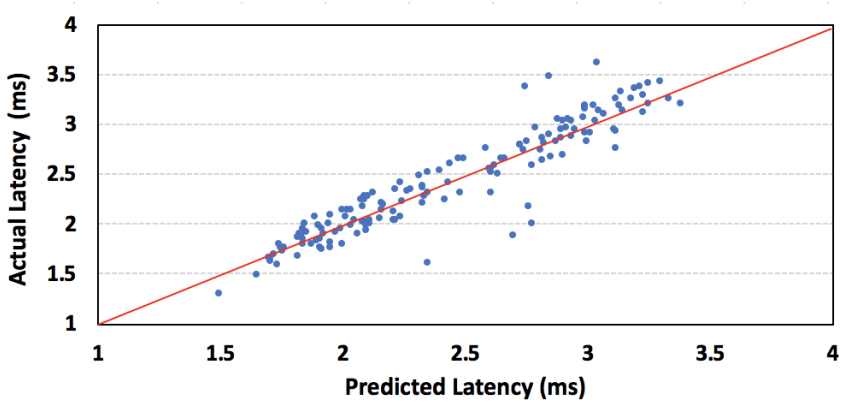


Figure 4.3. Effectiveness of kNN-based end-to-end latency predictor on the ImageNet data with 4 Tesla K80 GPUs. We determine $K = 5$ using cross-validation, which yields RMSE $0.22ms$ on the test set.

4.4.3 Results of Latency-aware NAS

Cifar-10 benchmark

After the architecture is searched, we train the weight parameters of the resulting network for 1,500 epochs using a batch size of 96 and Adam optimizer. As shown in Table 4.1, ColocNAS maintains a competitive test error rate (2.36% on average) in different hardware settings. In the meantime, ColocNAS reduces the inference latency by a large margin (1.24×/1.34×/1.50× faster on 2/3/4 Tesla K80 GPUs compared to DARTS 2-gpu) on multiple devices thanks to the parallelizable search space and differentiable latency guidance. ColocNAS uses a search space with elaborated connectivity and an online kNN-based latency lookup, thus the search cost is higher than DARTS (Table 4.1). Empirical results show that ColocNAS is still among the fastest NAS techniques due to our gradient-based method and CIFAR-10 proxy.

Table 4.1. Performance comparison between ColocNAS and the state-of-the-art NAS methods classification. The test error and inference latency on CIFAR-10 benchmarks are shown here. For the cell-based method, the latency is tested by reconstructing the model using open-source code from DARTS [94]. '-' indicates the model for CIFAR-10 is not publicly available. We use a batch size of 32 for all latency evaluations. The latency of ColocNAS is tested on 2/3/4 Tesla K80 GPUs after RL fine-tuning and 'DARTS 2-gpu' are tested on 2 GPUs using RL placement. Other baselines are tested on a single Tesla K80 GPU.

Model	Search Space	Search Method	Search Cost (GPU hours)	# Params	Test Error (%)	Latency (per image)	Normalized Reduction (%)
DenseNet-BC	-	manual	-	25.6M	3.46	14.8 ms	0.0
NAONet	cell	gradient	4.8K	10.6M	3.18	8.4 ms	-43.2
PNAS	cell	SMBO	6K	3.2M	3.41	3.14 ms	-78.8
Amoeba-A [126]	cell	evolution	76K	3.2M	3.12	3.10 ms	-79.1
Amoeba-B [126]	cell	evolution	76K	2.8M	2.55	3.04 ms	-79.4
DARTS (2nd) [94]	cell	gradient	24	3.2M	2.76	3.12 ms	-78.9
NASNet-A [186]	cell	RL	48K	3.3M	2.65	3.24 ms	-78.1
DARTS 2-gpu [94]	cell	gradient	24	3.2M	2.76	3.01 ms	-79.7
ColocNAS-b4c2	new space	gradient	41	3.4M	**2.31**	**2.43 ms**	**-83.6**
ColocNAS-b6c3	new space	gradient	49	3.5M	**2.35**	**2.24 ms**	**-84.9**
ColocNAS-b4c4	new space	gradient	51	3.4M	**2.42**	**2.01 ms**	**-86.4**

Table 4.2. Transferability classification error on ImageNet benchmarks. For NAS in layer-wise search space, the latency is tested by using open-source evaluation code from the papers. '-' indicates the number is not shown in the original paper or the model is not publicly available. The latency results in the table are tested on Tesla K80 GPUs. The RL setting for the baseline is evaluated on 2-GPU. With more GPUs, the baseline would yield the same latency as the models can hardly be parallelized across devices.

Model	Search Space	Search Method	Search Cost (GPU hours)	# Params	# FLOPs	Test Accuracy (Top-1 %)	Latency (-RL/+RL)	Normalized Reduction (%)
MobileNetV2	-	-	-	3.4M	300M	72.0	3.51/3.51 ms	0.0
ShuffleNetV2(1.5)	-	-	-	3.5M	299M	72.6	3.32/3.32 ms	-5.4
Amoeba-A [126]	cell	evolution	756K	5.1M	555M	74.5	3.13/ 3.02 ms	-14.0
NASNet-A [186]	cell	RL	48K	5.3M	564M	74.0	3.22/ 3.10 ms	-11.7
DARTS (2nd) [94]	cell	gradient	24	4.9M	595M	73.1	3.09/2.98 ms	-15.1
MnasNet-65 [147]	layer-wise	RL	91K	3.6M	270M	73.0	-	-
MnasNet [147]	layer-wise	RL	91K	4.2M	317M	74.0	-	-
FBNet-A [164]	layer-wise	gradient	216	4.3M	249M	73.0	2.93/2.93 ms	-16.5
FBNet-B [164]	layer-wise	gradient	216	4.5M	295M	74.1	3.41/3.41 ms	- 2.8
ProxylessNAS-mobile [17]	layer-wise	gradient	200	6.9M	-	**74.6**	3.95/3.95 ms	+12.5
ProxylessNAS-GPU [17]	layer-wise	gradient	200	7.1M	-	**75.1**	2.80/2.80 ms	-20.2
ColocNAS-b4c2	new space	gradient	41	4.7M	546M	**74.9**	**2.67/2.51 ms**	**-28.5**
ColocNAS-b6c3	new space	gradient	49	4.8M	572M	**74.6**	**2.41/2.23 ms**	**-36.5**
ColocNAS-b4c4	new space	gradient	51	4.8M	566M	**74.1**	**2.18/2.07 ms**	**-41.0**

ImageNet Results

Using the normal and reduction cell found by the latency-aware searching step, we stack 14 cells to build a neural network for ImageNet evaluation. The network is trained for 360 epochs with power cosine learning rate and SGD optimizer with Nesterov-momentum.

The comparison results are shown in Table 4.2. Compared to the state-of-the-art NAS methods, our model shows a competitive error rate on the test dataset (25.47% on average). Furthermore, our model outperforms all other state-of-the-art NAS methods in terms of inference latency when deploying in a multiple-device hardware environment. With 2/3/4 GPU settings, ColocNAS achieves 1.57×/1.77×/1.91× execution speedup compared to the ProxylessNAS-mobile on Tesla K80. For Tesla V100 GPUs, ColocNAS yields a similar speedup compared to DARTS and ProxylessNAS. Applying the RL placement, we directly tested models searched for Tesla K80 onto the new hardware systems and the latency per image for ColocNAS-b4c2/b6c3/b4c4 are 0.66/0.61/0.54 ms on 2/3/4 Tesla V100 GPUs, which are 1.12×/1.21×/1.35× faster than DARTS (0.74 ms) and 1.28×/1.39×/1.57× faster than ProxylessNAS-mobile (0.85

ms). This corroborates that ColocNAS preserves a high task accuracy while reducing the latency overhead. We also notice that if without the guidance of the hardware-bounded latency estimation ($\lambda_1 = 0$ in Eq. (4.7)), the latency incurred by ColocNAS will increase by +0.23ms (ColocNAS-b4c4) on Tesla K80 GPUs. Based on our observation, the latency regularization term encourages computation-efficient operations on the timing critical paths in the model. In addition, it explicitly helps to reduce the number of connectivity across blocks compared to model searching without latency guidance.

Performance Discussion

One can observe the trade-off between accuracy and latency from Table 4.1 and 4.2. ColocNAS aims to achieve hardware-aware NAS for fast model inference by reducing the communication overhead between multiple computing platforms. However, the reduction in communication would incur accuracy degradation since the information exchange between sub-graphs for each device is reduced.

Table 4.1 and 4.2 show that the latency improvement of ColocNAS is sublinear with the number of computing devices. This is caused by the non-negligible communication overhead between devices and system control overhead.

4.5 Conclusion

In this chapter, we propose ColocNAS, an end-to-end, differentiable neural architecture searching framework that is aware of the hardware-bounded latency. ColocNAS introduces an innovative parallelizable search space that reduces synchronization/device communication for model parallelism. For the first time, ColocNAS is able to maximize device utilization of multiple computing platforms by decoupling the task of hardware-aware NAS into three steps: offline hardware latency profiling, online latency-aware searching, and RL-based device placement fine-tuning. As a result, ColocNAS automates the design of neural architectures that achieve high task accuracy and low runtime latency for a given hardware setting. We perform extensive

experiments and corroborate that ColocNAS reduces the inference latency by a large margin in a multi-device environment while maintaining a competitive task accuracy compared to the state-of-the-art hardware-aware NAS methods.

Acknowledgement. Chapter 4, in part, contains a re-organized reprint of the material as it appears in IEEE Micro Journal 2020. Cheng Fu; Huili Chen; Zhenheng Yang; Farinaz Koushanfar; Yuandong Tian; Jishen Zhao. The dissertation author was the primary investigator and author of this paper.

Chapter 5

Accelerating DNN Training and Searching by Reusing Pretrained Model and Progressive Learning

5.1 Introduction

Vision transformer (ViT) models are promising to achieve the state-of-the-art performance on various computer vision tasks [39, 52, 151, 152]. While the performance frontier keeps pushing forward, the training costs also scale with the growth of parameters. For example, Dehghani et al. [35] scaled a ViT to 22 billion parameters. Furthermore, recent research [99] shows that transformers require much more training steps and larger datasets to better generalize compared to convolutional neural networks (CNNs), imposing even more scaling costs of ViTs.

Among various transformer training acceleration and cost reduction techniques [67, 25, 166, 175, 140], one promising method is to reuse small pretrained models to initialize a large model before training. By scaling a small pretrained model with various expansion operators and using it to initialize the large model, the implicit knowledge facilitates faster model convergence. Previous studies such as bert2BERT [21] focused on preserving the functionality of the small pretrained transformer, when growing transformer width (i.e., hidden dimensions). A recent work learn-to-grow [159] learns linear mappings to scale the pretrained model by minimizing the task loss during model expansion.

However, we identify a set of critical limitations in the existing approaches through a comprehensive investigation (Sec 5.2.3). (**L1**) Maintaining the functionality during model scaling is not the only key factor to achieving a high speedup and final task accuracy. We identify that the other critical factor is retaining the *optimizer states* of the weights, because that preserves the direction of model updates when the functionality is preserved. Previous methods [159, 21] do not include these optimizer states during model scaling. (**L2**) Training discrepancies exist between the pretrained weights and the new weights because the new weights introduced during scaling do not have optimizer states built up before training. (**L3**) The functionality of a pretrained model is hardly preserved by simply expanding a small pretrained model in multiple dimensions all at once.

To address these limitations, we propose a new method, TripLe[1], which *partially expands a pretrained model before training* and *grows the rest of the parameters during training.* TripLe conducts the expansion of model width and depth in a serialized fashion. Tackling **L1**, we scale the width of the pretrained transformer with their optimizer states to initialize the scaled model and optimizer. So the pretrained weights will maintain their updated directions during training. After a short warmup training phase, the new weights will obtain their training states. To address **L2**, we increase the depth of ViTs by copying the existing weight parameters and their training states. As such, the optimizer states obtained from the first stage can be leveraged by the second expansion, mitigating the training mismatch between new and pretrained weights. For **L3**, when serializing the width and depth expansion, each expansion stage will mostly preserve the functionality of the small pretrained model, enabling faster convergence of the training period.

TripLe offers the best of the two worlds – *pretrained model reuse* and *progressive learning*. Our exploration shows that they are two extreme cases for transformer scaling: the pretrained model reuse technique will scale a small pretrained model in multiple dimensions toward the target model before training, while progressive learning starts from a randomly initialized model

[1]We call it TripLe because it incorporates three principles: reusing pretrained models, progressive learning, and knowledge distillation.

and grows parameters during training until reaching the target large model. We observe that these methods in fact benefit each other on ViT training: reusing a pretrained ViT facilitates faster convergence of progressive learning at each stage, while progressive learning constructs the training states that help the ViT scaling. To further improve model quality, we augment TripLe with knowledge distillation (KD) [60]. Specifically, TripLe applies the pretrained model to initialize a large model and KD uses the pretrained model to provide teaching signals during training.

Experiments and Results. We evaluate TripLe with both *single-trial model scaling* and *multi-trial neural architecture search* (NAS). With single-trial training, we scale various ViTs, and compare the training time and task accuracy against from-scratch training and a variety of baseline model scaling methods. When scaling pretrained ViTs size by 8×, TripLe saves the training time up to 71.0%∼80.9% compared to from-scratch training. Other model scaling methods hardly achieve performance neutrality against from-scratch training on large ViT models. Moreover, with the same training budget as from-scratch training, a 44MB ViT (expanded from a 5MB ViT) outperforms both the official 86MB DeiT-B (by 0.2%) and KD alone (by 1.0%) in ImageNet-1k task accuracy. Combining TripLe with KD, the 44MB model shows a 1.8% higher task accuracy compared to using KD alone.

TripLe enhances NAS performance by finding a 27MB model that outperforms the task accuracy of the 86MB DeiT-B [151] and the model searched by traditional multi-trial NAS. One of the significant downsides of multi-trial search is a long searching time [185], as sampled models are always trained from scratch. To reduce training time, prior approaches typically use ∼10% of the final training time to approximate model accuracy. Yet, this proxy accuracy is too far from the final accuracy. Our method can be viewed as a new form of 'weight-sharing' designated for multi-trial NAS. TripLe allows each trial to start from a pretrained model (Figure 5.2). The proxy accuracy obtained from TripLe shows a higher correlation with the final task accuracy compared to training from scratch. After the model is searched, we further improve the model performance using TripLe during model evaluation.

Table 5.1. ViT [151] for evaluating TripLe. New model variants S_{L24}/B_{L24} only change the #layers of DeiT-S/DeiT-B.

Model (DeiT-)	hidden dim	#heads	#layers	#params	FLOPs (billions)	Top-1† Acc	Top-1‡ Acc
Ti	192	3	12	5M	2.16	72.0	72.1
S	384	6	12	22M	8.50	79.5	79.8
B	768	12	12	86M	33.72	81.0	81.8
L	1024	16	24	307M	119.36	82.1	82.2
S_{L24}	384	6	24	44M	16.87	80.0	-
B_{L24}	768	12	24	172M	67.21	81.4	-

† Top-1 Accuracy of our DeiT re-implementation with 300 epochs of training.
‡ Official Top-1 task accuracy under 300 epochs of training.

5.2 Scaling Vision Transformers

In this section, we introduce the Vision Transformers (ViTs) studied in this section. Furthermore, we perform a comprehensive investigation on the performance of existing scaling operators.

5.2.1 Vision Transformer

Transformer [38] was first employed in vision tasks by Dosovitskiy et al. [39]. ViT first extracts features from raw image patches using a CNN and feeds the extracted features as the input to the transformer. DeiT [151] finds that the model can achieve a high task accuracy on ImageNet-1k [36] dataset when applying strong image augmentation with knowledge distillation [4]. Many follow-up works propose ViT variants for better task accuracy [152, 52, 153, 181, 22].

This study focuses on scaling pretrained ViTs, and our experiments reuse the DeiT architectures [151]. To study scaling ViTs in both depth and width, we also introduce two model variants namely DeiT-B_{L24} and DeiT-S_{L24} given in Table 5.1.

5.2.2 Revisiting Operators Scaling

Recent studies [21, 159] reuse small pretrained transformers to accelerate the large model training. These works introduce different expansion operators for transformer depth (i.e., number

of layers) and width (i.e., hidden dimension). They then employ the expanded pretrained model to initialize the target model. We denote the width/depth expansion operator as γ/β. The large model Θ to be trained has L transformer layers with hidden dimension D. The pretrained model θ has l layers with hidden dimension d.

Layer Stacking β_{stck} and Interpolation β_{inpt}. Layer stacking [50] and interpolation [20] are two common approaches to increasing the transformer depth. Specifically, the operation can be formulated as follows:

$$\beta_{stck} : W_i = w_{i \mod l}, \ \ \forall i \in \{1, ..., L\} \tag{5.1}$$

$$\beta_{inpt} : W_i = w_{\lfloor i/k \rfloor}, \ \ \forall i \in \{1, ..., L\}, k = \lfloor L/l \rfloor \tag{5.2}$$

Here, W_i denotes the initial weight of the i-th transformer layer in model Θ. k is the expansion ratio of layers. By duplicating the existing layers according to Eq.5.1-5.2, we can scale the pretrained model in depth.

Adding Identity Layers β_{ST}. Shen et al.[139] propose to add identity layers W_I to maintain the functionality of the pretrained model. We denote this layer as W_I where $W_I(x) = x$. Specifically, each transformer layer can be viewed as two sub-layers:

$$\begin{aligned} x' &= x + \text{Attention}(LN(x)) \\ y &= x' + \text{FFN}(LN(x')) \end{aligned} \tag{5.3}$$

The 'Attention' denotes the multi-head attention layer. 'FFN' denotes the feed-forward layers following the attention layer [155]. 'LN' denotes the layer normalization. When initializing the scale and bias in 'LN', 'Attention', and 'FFN' to 0, the output of Attention$(LN(x))$ and FFN$(LN(x'))$ will be 0 as well. In this way, the transformer layer has y and x equal. β_{ST} can be combined with layer β_{stck} or β_{inpt}, i.e., where to add these identity layers. They are denoted as β_{STstck} and β_{STinpt}, respectively.

bert2BERT γ_{b2B}: Net2Net [23] increases the width of neural networks by duplicating neurons randomly and maintaining their output values through normalization. For transformers, it is first applied in bert2BERT [21] for pretrained transformer scaling (Detailed in Appendix B.2).

Padding Zeros γ_{pad0}: A straightforward way to increase the transformer width is to pad zeros to the existing weights. The small pretrained weights are on the upper-left corner of the large layer, the rest parameters are all zero-initialized.

Special padding zeros γ_{ST}: Staged-training [139] proposes a new width expansion method. When scaling a dense layer, the width expansion can be written as:

$$\gamma_{ST}(w) = \begin{pmatrix} w & z \\ z & w \end{pmatrix} \tag{5.4}$$

Here, z is a $d \times d$ zero matrix. The scaled matrix has a size of $D \times D$, where $D = 2d$. (Equations for other parameters are detailed in Appendix B.2.)

Weight resizing γ_{inpt}: In this work, we propose a new baseline operator interpolation γ_{inpt} that treats the weight matrices as images and interpolates the matrices using image resizing methods, such as bicubic / bilinear [34] and etc. Empirically, we find bilinear outperforms other methods, so we apply it in γ_{inpt}.

Learn-to-grow γ_{lgt} / β_{lgt}: Besides all the above methods, learn-to-grow [159] proposes to learn linear matrices that map the pretrained weights into larger weight matrices to preserve the functionality of the small pretrained model (Equations in Appendix B.2.). We denote its width and depth expansion operator as γ_{ltg} and β_{ltg}, respectively.

The combination of these operators forms all the existing methods for scaling pretrained transformers. The summary is given in Table 5.2.

Table 5.2. Relationships between different methods and expansion operators for different transformer scaling methods. Model Interpolation is a new baseline proposed in this section.

Method	Notation	width	depth
bert2BERT [21]	b2B	γ_{b2B}	β_{stck}
Staged-Training [139]	ST	γ_{ST}	β_{STinpt}
Model Interpolation	Inpt	γ_{inpt}	β_{inpt}
Learn-to-grow [159]	LTG	γ_{lgt}	β_{lgt}
Pad Zero	Pad0	γ_{pad0}	β_{stck}

5.2.3 Investigating Expansion Operators

We motivate TripLe with a detailed investigation of the performance and inefficiencies of previous scaling operators.

Investigation 1: Which operators can preserve model functionality? Many different expansion methods, such as γ_{b2B}, γ_{ST} and β_{ST} claim they are functionality preserving. We rebuild these baselines and their initialized accuracy is given in Table 5.3 (marked in blue).

① **bert2BERT** (γ_{b2B}): We find γ_{b2B} can preserve transformer functionality under constraint. When applying γ_{b2B} to scale a $d \times d$ dense layer into $2d \times 2d$, the original output vector o can become $\bar{o} = \{\frac{o}{2}, \frac{o}{2}\}$. After another non-linear function F, the output $F(\bar{o})$ can be recovered back to $F(o)$ through another linear function when F satisfies:

$$F(x) = F(x/n) \cdot n, \qquad n \in R, x \in R \tag{5.5}$$

As GeLU in ViT doesn't satisfy Eq.5.5, the functionality cannot be preserved. When switching GeLU to ReLU in FFN, we find the ViT functionality can be fully maintained using γ_{b2B} (Appendix B.2).

② γ_{ST} **and** β_{ST}: We find the operator γ_{ST}/β_{ST} can preserve the functionality of the ViT. β_{ST} is an identity layer and it is functionality preserving as discussed in Sec 5.2.2. For γ_{ST}, when expanding a $d \times d$ dense layer into $2d \times 2d$, The new dense layer has output $\bar{o} = \{o, o\}$ where o

is the original output with a size of $d \times 1$ according to Eq.5.4. Because γ_{ST} does not scale o as γ_{b2B}, the output results GeLU($\bar{o}$) can be recovered back to GeLU(o) after another linear mapping.

③ β_{stck} **and** β_{inpt}: Empirically, we also find β_{stck} and β_{inpt} can preserve partial functionality. For example, when expanding S→S$_{L24}$, the expanded model with β_{stck}/β_{inpt} can achieve 65.56%/39.98% task accuracy, respectively. This indicates the initial task loss is small after scaling the pretrained model using β_{stck} / β_{inpt}.

④ **Learn-to-grow** γ_{ltg} **and** β_{ltg}: Learn-to-grow focuses on reducing the task loss L at the beginning of the training as given in Eq.5.6 through learning linear mappings (β_{ltg}, γ_{ltg}) from the small pretrained model to the large one.

$$\underset{\beta_{ltg}\gamma_{ltg}}{\arg\min} E_{x\sim Dt} L(x;\Theta), \quad s.t. \Theta = \beta_{ltg}(\gamma_{ltg}(\theta)) \tag{5.6}$$

Here, Dt represents the data distribution, and Θ is a large model expanded from a small checkpoint θ. When expanding S→B, learn-to-grow can only achieve 72% initial task accuracy [159] compared to 79.5% task accuracy of the small pretrained model. As such, we conclude that γ_{lgt}/β_{lgt} can only preserve partial functionality.

Other scaling operators discussed in Sec. 5.2.2 are not functionally preserving.

Investigation 2: Does the initial accuracy of the scaled model matter to the final task performance?

We observe that the correlation between the initial accuracy of the scaled model and the final accuracy is weak. Specifically, we evaluate each width/depth operator using the ImageNet-1k dataset. (Hyperparameters in Appendix A). The results are given in Table 5.3 (marked the columns in purple).

Among width expansion operators, γ_{ST} achieves the highest initial accuracy, however, it cannot achieve the best final accuracy across the baselines. The initial accuracy obtained using γ_{ST} can be compromised within a few training iterations. Our new baseline γ_{inpt} (not

Table 5.3. Performance comparison between different expansion operators under 30/60 training epochs (ep_{30}/ep_{60}). '-m' denotes ignoring the optimizer states. '+m' means we scale the optimizer states and use them to initialize the new optimizer.

models	width γ	depth β	Test accuracy					Δ_{ep60}
			init	-m		+m		
				ep30	ep60	ep30	ep60	
Ti→S	γ_{b2B}	-	0.00	72.90	**76.53**	72.76	76.64	+0.11
Ti→S	γ_{ST}	-	71.82	72.82	75.72	**74.88**	**77.29**	**+1.57**
Ti→S	γ_{pad0}	-	0.01	69.89	70.83	69.85	72.45	**+1.63**
Ti→S	γ_{inpt}	-	0.00	**73.04**	76.50	73.25	76.11	-0.39
S→S_{L24}	-	β_{stck}	65.56	79.10	80.74	80.02	**81.31**	**+0.57**
S→S_{L24}	-	β_{inpt}	38.98	**80.00**	**81.23**	**80.34**	81.26	+0.03
S→S_{L24}	-	β_{STstck}	79.49	78.46	79.47	78.67	79.52	+0.05
S→S_{L24}	-	β_{STinpt}	79.49	78.61	79.23	78.41	79.26	+0.03
Ti→S_{L24}	γ_{ST}	β_{stck}	0.01	75.73	78.21	76.23	78.62	+0.40
Ti→S_{L24}	TripLe		71.82	-	-	**76.52**	**79.23**	-

functionality preserving) achieves very similar final task accuracy compared to other baselines. For depth expansion operator, γ_{STstck}/γ_{STinpt}, can maintain model functionality. Yet, the identity transformer layer (Eq.5.3) poses a considerable challenge for the model to be trained properly.

Investigation 3: Effect of optimizer states in scaling pretrained ViTs. ViTs are mostly trained using AdamW [100]. That means the optimizer states also exist in the pretrained model. Specifically, during the model update, we have:

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t \tag{5.7}$$

$$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2 \tag{5.8}$$

$$\theta_{t+1} = \theta_t - \frac{\lambda(t)}{\sqrt{v_t}+\varepsilon} \cdot m_t \tag{5.9}$$

Here, g_t is the first-order gradient of the weight parameter θ at time step t. $\lambda(t)$ is the learning rate scheduler. β_1 and β_2 are constant decay rates (0.9, 0.999). m and v are AdamW states with the same dimensions as θ. Previous work, such as learn-to-grow or bert2BERT neglects this momentum information.

To investigate whether m and v help reduce the training time, we also apply the same width and depth operators γ/β on m and v during model initialization. The performance differences are listed in Table 5.3 (marked in red).

The results show that retaining optimizer states improves the model quality in general, especially when the functionality of the pretrained model is preserved or at least partially preserved. With width expansion, the functionality preserving γ_{ST} with momentum information outperforms the second-best baseline by 0.65%. With depth expansion, β_{stck} shows a 0.57% accuracy increase with momentum information. This is because functionality preserving reduces the initial task loss and maintains the gradient g_t in Eq.5.7-5.8 for pretrained weights. Together with m_{t-1} and v_{t-1}, the direction of pretrained weights update $\frac{m_t}{\sqrt{v_t}+\varepsilon}$ is preserved. One exception is β_{ST}, because g_t in the growing layers changes due to zero-initialized LN and bias (Eq.5.3).

Investigation 4: Can we perfectly scale Ti to S_{L24} by combining the best scaling operators among γ and β? We expand the model from Ti$\rightarrow$$S_{L24}$ using the best-performed operator selected from the previous investigation, namely, γ_{ST} and β_{stck} (Table 5.3). Ti$\rightarrow$$S_{L24}$ shows a 2.7% accuracy drop compared to S$\rightarrow$$S_{L24}$. This is due to the limitations of simply combining γ_{ST} and β_{stck}. (1) The functionality-preserving feature of γ_{ST} and β_{stck} is compromised, when combining them together. This incurs the advantage of applying optimizer states. (2) For Ti$\rightarrow$$S_{L24}$, the weights grow from 5MB to 44MB. Among them, 24MB of the weights are *zeros* introduced during model expansion. These zero values have no training states in the pretrained model. In addition, zero values are under-trained. As such, a training discrepancy exists between the zeros and the pretrained parameters. The issue is exacerbated once weight decay is applied because the zero values will not be penalized at the beginning of training.

5.3 TripLe for Scaling Single Model

Overview. We propose a new method to mitigate the aforementioned training mismatches by serializing the model expansion. As discussed above, simply combining the best-performed scaling operators β_{stck} and γ_{ST} together is sub-optimal. We propose TripLe that *scales width before training* and *grows model depth during training* (Figure 5.1(a)). TripLe maintains the functionality preserving feature of γ_{ST} and β_{stck} by serializing the scaling operations. Furthermore, with a short warmup training after conducting γ_{ST}, zero weights will obtain their training states that benefit the subsequent depth expansion. In what follows, we discuss each step in detail.

Scaling Width Before Training. TripLe applies γ_{ST} to expand transformer width before the training. We extend γ_{ST} to the scenario where the expanded model width is not divisible by the small model width (Eq.5.10). This method is for more general model scaling (e.g., NAS).

$$\gamma_{ST}(w) = k\left\{\begin{bmatrix} w & \dots & z & z_s \\ \vdots & \ddots & \vdots & \vdots \\ z & \dots & w & z_s \\ z_s^T & \dots & z_s^T & w_s \end{bmatrix}\right. ,k = \lfloor D/d \rfloor \tag{5.10}$$

Here, w is the $d \times d$ pretrained dense layer. w_s is downsampled from w with a size of $ds \times ds$, $ds = D \mod d$. z and z_s are zero matrices with different sizes. The scaled layer $\gamma_{ST}(w)$ has a size of $D \times D$. For parameters b with a dimension of $1 \times d$ in LN, weight bias, and classification token, we can conduct a similar procedure using Eq.5.11.

$$\gamma_{ST}(b) = \left(\overbrace{\begin{matrix} b & \dots & b \end{matrix}}^{k} \; b_s\right) ,k = \lfloor D/d \rfloor \tag{5.11}$$

Here, '$(\cdot)$' is the concatenation operation, $\gamma_{ST}(b)$ has a size of $1 \times D$. As m and v have the same dimension as their weight parameters, we apply the same operators (Eq.5.10-5.11) on m/v

and use $\gamma_{ST}(m)/\gamma_{ST}(v)$ to initialize the AdamW optimizer. For the CNN in ViT with a dimension of $(d, \text{channel}, r, s)$, we flatten it along kernel dimension (d) and apply Eq.5.11. The CNN after scaling will be reshaped back to $(D, \text{channel}, r, s)$.

Empirically, we find this method can maintain partial functionality at the beginning of the training when D is not divisible by d.

Growing Model Depth During Training. As shown in Figure 5.1, before growing the depth of the model, we conduct warm-up training by keeping the pretrained model depth (Stage I). After Stage I, we directly copy-paste the weight parameters and optimizer states from the bottom transformer layers to the top layers. The number of layers will grow from l to $L = 2l$. In this stage (Stage II), we freeze the bottom l layers including the positional encoding and convolution layer. Lastly (Stage III), we unfreeze the bottom layer and train all the weights together.

The key difference between our approach and progressive learning[166] is that: (1) besides the weight parameters, we also copy the momentum information which is effective in speedup the training (Sec 5.2.3) (2) Traditional progressive learning requires a long training time for each stage to converge. However, when reusing a pretrained model, each stage can converge in a very short amount of time.

TripLe without Depth Expansion. When the depth of the small pretrained model and the target model is the same (l), we cannot copy-paste the warmed-up parameters from the bottom layer i ($i \in \{1, .., \frac{l}{2}\}$) to the top layer j ($j \in \{\frac{l}{2} + 1, .., l\}$), as the top layers have already been initialized with pretrained weights. As discussed in Sec 5.2.3, zero weights (z) are under-trained compared to pretrained weight (w) after conducting γ_{ST} (Eq.5.10). As such, we warm up the bottom layers and the zeros at the top layers in Stage I (Figure 5.1(b)). Next in Stage II, we train only the matrices that are zero-initialized in the bottom layers (position z/z_s in Eq.5.10) and all the parameters in the top layers. In this way, zero weights have more training steps compared to the pretrained weights during warm-up stages, mitigating the problem that zeros are under-trained (Sec 5.2.3). Also, the training states for zero weights are established before training all parameters together. Empirically, we find this method improves the training speed

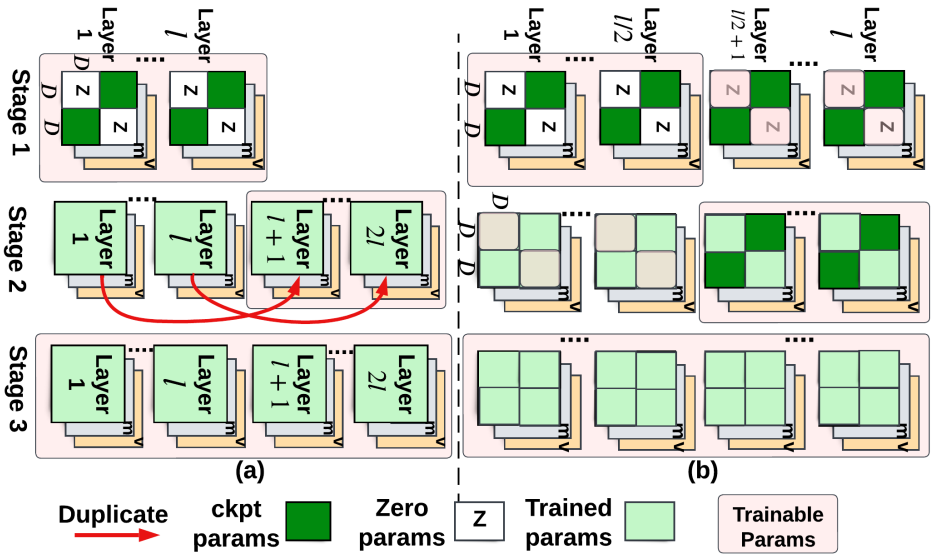


Figure 5.1. Training stages in TripLe when the model depth (l) (a) scales by $2\times$ and (b) is not scaling. We simplify the transformer layer into a single MLP. All the dense layers in the transformer layer are operated in the same fashion.

and can achieve better task performance when scaling width only.

Combining TripLe with Knowledge Distillation. We also find *reusing pretrained models* methods can be combined with *Knowledge Distillation* (KD) to further improve the model performance. For KD, the pretrained models are employed to provide training signals; for methods in reusing pretrained models, the pretrained weights are used to initialize the large model. Based on our knowledge, we are the first work to combine them together.

We follow the KD method introduced in DeiT [151] and use the 'hard-label distillation' loss during training. Since our architectures are the same as DeiT, the integration is straightforward (Detailed in Appendix C.2). To make a fair comparison with the previous methods, we do not add KD during training unless specified.

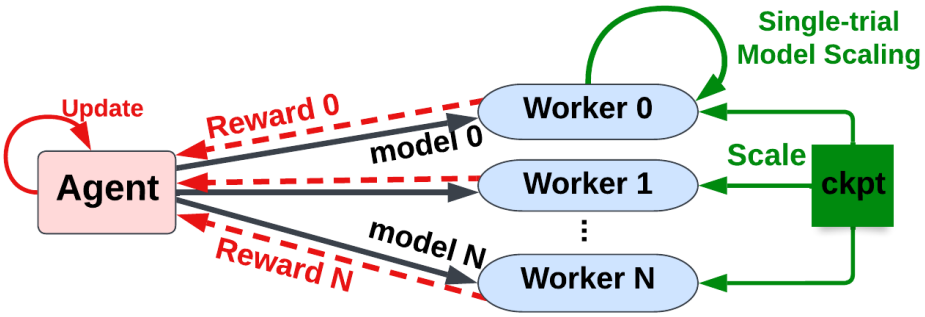


Figure 5.2. Leveraging TripLe in multi-trial NAS. The green blocks and arrows are key differences compared to traditional multi-trial NAS, while 'ckpt' denotes the small pretrained model.

5.4 TripLe for Multi-trial NAS

As one of use cases of model scaling is to enhance multi-trial NAS, we design a ViT search space and evaluate TripLe against traditional approaches as shown in Figure 5.2.

Traditional multi-trial NAS adopts an agent to sample a model and training hyperparameters from the search space. A worker starts training the model from scratch based on this selection. Different from traditional NAS, the worker in our approach will start training from a pretrained model leveraging TripLe. This section describes our search space, searching algorithm, and reward function.

Multi-trial Search Space. We build a neural architecture search space based on ViT architecture (Table 5.4). We search head factors h_f of each layer, hidden dimensions D, number of layers, and FFN expansion ratio e_f. The number of heads of each layer is D/h_f. The dimension of the FFN layer is $D \times e_f$. D, h_f and e_f are sampled independently. Furthermore, we search the learning rate and weight decay, which cannot be explored using one-shot algorithms. The cardinality of our search space is around 9.4e11.

Searching Algorithm and Reward function. We apply a regularized evolution algorithm [126] as the controller algorithm to optimize the search space of NAS. We do not choose to employ PPO method [134, 147, 185, 148] which requires a long training time for the agent to converge.

Table 5.4. ViT search space for evaluating TripLe in NAS. 'Global' means all the layers are set to the same sampled parameter. 'Local' means the parameters are sampled for each layer.

Parameter name	Selections	Global/Local
hidden dimension (D)	[192, 384, 576, 768]	Global
Head factor (h_f)	[32, 64]	Local
FFN Expansion factor (e_f)	[2, 3, 4]	Local
Transformer layers (L)	[12, 13,..., 24]	Global
Learning rate (λ)	[5e-4, 1e-3, 4e-3]	Global
Weight decay	[0.05, 0.02, 0]	Global

We adopt the TuNAS reward [11] and our reward is defined as

$$Reward(\Theta) = Q(\Theta) + \varepsilon|\frac{FLOPs(\Theta)}{FLOPs_0} - 1| \tag{5.12}$$

Here, $Q(\cdot)$ indicates the quality (accuracy) of a candidate architecture Θ, $FLOPs(\Theta)$ is its FLOPs, $FLOPs_0$ is a problem-dependent FLOPs target, and hyperparameter $\varepsilon < 0$ is the cost exponent.

5.5 Evaluation

We evaluate the performance of TripLe in both single-trial model scaling and multi-trial NAS.

Dataset and Hyperparameters. We evaluate TripLe using ImageNet-1k [36] for training ViTs. The ViT architectures are given in Table 5.1. We transfer the models trained using TripLe to various downstream tasks which include CIFAR10 [81], CIFAR100 [81], Flowers102 [109], StanfordCars [80] (results of given in Appendix F). All the experiments are done on dragonfish TPUs [74] with 8×8 topology. The validation/test sets are evaluated every 400 seconds on separate TPUs.

We set the batch size to 4096, so the learning rate would be $\frac{batchsize}{512} * 0.0005 = 0.004$ according to the DeiT paper. Other hyperparameters are the same as DeiT [151] (Detailed in Appendix A.1). Our baseline methods are given in Table 5.3. For ST and Inpt, we also initialize the scaled model with optimizer states for a fair comparison. Specifically, we conduct the width/depth expansion on m and v using the same operators to expand the weights. MLST [166] is a progressive learning baseline for transformer training.

We set the training time t for each model scaling task to 30/60/90/120/300 epochs (denoted as ep_t), respectively. The final learning rates under different training times are always 0. The warm-up epochs are set to 5. For TripLe, the depth growth happens during the warm-up phase and Stage1/Stage2 takes 2.5/2.5 epochs, respectively.

5.5.1 Evaluation of Single-trial Models Scaling

Metrics. For evaluating the model quality, we report the Top-1 test accuracy on ImageNet-1k.

To measure the training cost for each method, we scan the minimum time required to match the *validation loss* of from-scratch training. And then, we use it to compute the maximum wall-time reduction **(Max ↓ Time)** and maximum training FLOPs reduction **(Max ↓ FLOPs)** for each method accordingly. When the method is unable to achieve the validation loss of

Table 5.5. Performance comparison between different model scaling methods. ep_{30} denotes the total training time is set to 30 epochs.

Model	Max ↓ Time	Method	Max ↓ FLOPs	Top-1 Test accuracy (%)				
				ep_{30}	ep_{60}	ep_{90}	ep_{120}	ep_{300}
B	0%	Scratch	0%	-	-	-	-	81.03
B	28.9%	MLST	36.7%	-	-	-	-	81.27
S→B	52.0%	LTG	55.4%	-	-	-	-	81.57
S→B	67.7%	b2B	68.2%	78.11	80.39	81.45	81.82	81.81
S→B	74.2%	Inpt	74.8%	78.98	80.89	81.80	81.75	81.75
S→B	78.6%	ST	80.0%	79.33	81.29	81.99	82.10	81.92
S→B	**81.4%**	TripLe	**82.1%**	**79.72**	**81.32**	**82.10**	**82.36**	**82.01**
S	0%	Scratch	0%	-	-	-	-	79.50
S	×	MLST	×	-	-	-	-	75.97
Ti→S	12.0%	b2B	12.2%	72.76	76.64	78.01	79.01	80.33
Ti→S	11.2%	Inpt	11.5%	73.25	76.11	77.78	78.59	80.54
Ti→S	12.0%	ST	13.8%	**74.88**	77.29	78.46	79.25	80.67
Ti→S	**17.4%**	TripLe	**18.4%**	74.67	**77.53**	**78.54**	**79.34**	**80.88**

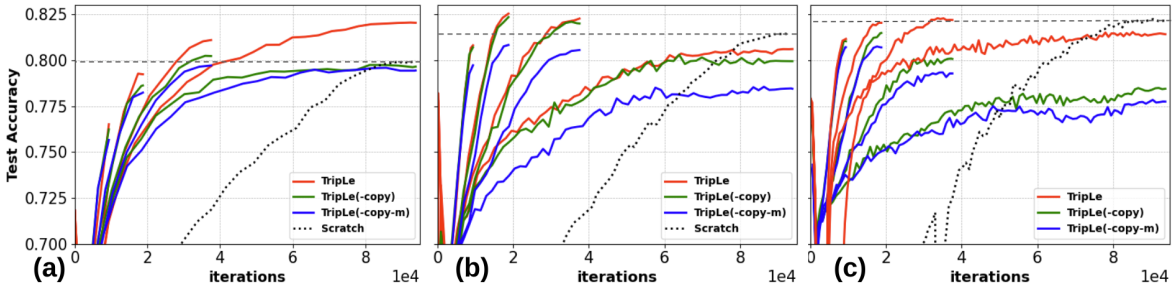
from-scratch training under any settings, we report '×' in the corresponding table entry.

Evaluation on Expanding Width Only. We first perform width scaling only to evaluate our method given in Figure 5.1(b). As is shown in Table 5.5, TripLe outperforms other baselines in max training time reduction. For Ti→S/S→B, TripLe can save the 17.4%/82.1% maximum training time compared to 12.0%/78.6% of ST (the second-best baseline).

Besides, TripLe can achieve better task accuracy compared to baselines generally. When using 300 epochs for training, TripLe can outperform from-scratch training accuracy by 1.38%/0.98% for Ti→S/S→B. This indicates that training more steps on the zero weights introduced by *ST* can mitigate the training mismatch between zero weights and pretrained weights. For learn-to-grow, the implementation is not open-source; so we report the max ↓ Time/FLOPs using the number reported in the paper.

Expanding Width and Depth Together. When expanding width and depth together, we apply our method given in Figure 5.1(a) that serializes the expansion of ViTs. The model will grow 8× in parameter size. When choosing 40% (ep_{120}) of the total training time (300 epochs), the model obtained from TripLe outperforms ST by 0.88%/0.27%/0.69% and scratch training by

Figure 5.3. Sensitivity analysis of TripLe for (a) Ti→S_{L24} (b) S→B_{L24} (c) B→L under ep_{30}/ep_{60}/ep_{120}/ep_{300}. '-copy' denotes scaling both width and depth together before training. '-m' denotes ignoring momentum information from the pretrained model.



1.03%/0.99%/0.09% in task accuracy. This shows that serializing expansion operators can obtain better task accuracy under the same training budget. Besides the saving training cost, TripLe can also be employed to train ViT for better task accuracy compared to training from scratch.

Also, using progressive learning alone (i.e., MLST) cannot reach task performance of scratch training for B_{L24}/L under ep_{300}. The existing weights cannot be fully trained before expanding to a larger model, resulting in performance degradation.

Comparison and Combination of TripLe with Knowledge Distillation. As discussed in Sec 5.3, TripLe is orthogonal to KD which is another widely used technique to improve the transformer quality. In this subsection, we compare and combine TripLe with KD. For the teacher model in KD, we use a ResNet-101 with 79.33% test accuracy. The ResNet-101 must be trained using the same data augmentation techniques as DeiT.

The learning curves of TripLe and KD are given Figure 5.4. We observe that using TripLe can outperform the model trained using KD. For Ti→S_{L24}, the model achieves 82.0% test accuracy compared to 81.0% obtained from KD. When combining TripLe with KD, the model accuracy of S_{L24} can reach 82.8% test accuracy. This combination reveals that not only can we use the pretrained model to provide teaching signals in KD, but we can also use the small pretrained model to initialize the large model directly.

Sensitivity Analysis of TripLe. We gradually remove the design components from TripLe to

Table 5.6. Performance comparison between different model scaling methods. ep_{30} denotes the total training time is set to 30 epochs. Ti→S_{L24} denotes scaling DeiT-Ti to DeiT-S_{L24}. The pretrained model accuracy is given in Table 5.1.

models	Max ↓ Time	Method	Max ↓ FLOPs	Top-1 Test accuracy (%)				
				ep_{30}	ep_{60}	ep_{90}	ep_{120}	ep_{300}
S_{L24}	0%	Scratch	0%	-	-	-	-	79.97
S_{L24}	39.3%	MLST	41.6%	-	-	-	-	80.42
Ti→ S_{L24}	58.9%	b2B	60.1%	75.28	78.63	79.98	80.14	80.54
Ti→ S_{L24}	68.6%	Inpt	70.1%	75.72	78.93	80.40	80.64	81.12
Ti→ S_{L24}	67.9%	ST	70.0%	76.23	78.62	80.10	80.22	79.65
Ti→ S_{L24}	**71.0%**	TripLe	**72.1%**	**76.52**	**79.23**	**80.68**	**81.10**	**82.04**
B_{L24}	0%	Scratch	0%	-	-	-	-	**81.37**
B_{L24}	×	MLST	×	-	-	-	-	78.84
S→ B_{L24}	×	b2B	×	79.02	80.89	81.33	81.22	79.39
S→ B_{L24}	79.9%	Inpt	80.4%	80.40	82.29	**82.39**	82.23	80.28
S→ B_{L24}	79.9%	ST	80.4%	80.70	82.34	82.23	81.99	79.94
S→ B_{L24}	**80.9%**	TripLe	**81.7%**	**80.77**	**82.53**	82.36	**82.26**	80.60
L	0%	Scratch	0%	-	-	-	-	**82.12**
L	×	MLST	×	-	-	-	-	49.63
B→ L	×	b2B	×	78.02	81.64	81.71	81.55	80.01
L	×	Inpt	×	81.23	81.55	81.28	81.20	80.59
B→ L	×	ST	×	80.88	81.45	81.65	81.50	79.84
B→ L	**73.3%**	TripLe	**74.7%**	**81.23**	**82.04**	**82.22**	**82.19**	81.40

validate their effects. The learning curves are given in Figure 5.3. When switching our depth expansion method to β_{stck} and conducting the expansion all at once before training (TripLe-copy), the performance gets worse and the model is overfitting in long-time training. This shows that conducting all the expansions before training incurs performance degradation. We further ignore the momentum information (TripLe-copy-m) during the model scaling and the results become even worse compared to the previous analysis. As such, maintaining the train states is critical for scaling pretrained models.

Sensitivity to Training Times. In Table 5.6, we present the task performance as a function of training time. Keeping the original training budget with model scaling methods incurs performance degradation. That is because scaling pretrained model achieves faster training convergence. In this scenario, training the large model for too long will result in model overfitting. For training from scratch, the overfitting doesn't occur until the model is trained for 400 epochs [151]. As such, reducing the training time when scaling a pretrained model is necessary.

Table 5.7. Kendall-tau correlation between different methods across 15 trials.

Method	Scratch$_{ep30}$	TripLe$_{ep30}$	Scratch$_{ep300}$
Scratch$_{ep300}$	0.221	**0.318**	-
TripLe$_{ep120}$	0.789	**0.865**	0.363

5.5.2 Evaluation of TripLe on Multi-trial Search

We intend to answer two questions in this section: (1) *Does the proxy accuracy obtained from TripLe show a higher correlation to the final task accuracy?* (2) *Can TripLe find better models compared to multi-trial search?*

The pretrained model we reuse for the sampled models is DeiT-Ti. Our searching method is implemented inside the symbolic programming library named PyGlove [113].

Ranking Score Comparison. We randomize 15 models from our search space (Table 5.4) and evaluate the Kendall-tau [78] correlation between the proxy accuracy and the task accuracy. Specifically, the models are trained: (1) from scratch for 30 epochs. (Scratch$_{ep30}$) (2) from the pre-trained model for 30 epochs. (TripLe$_{ep30}$) (3) from scratch for 300 epochs (Scratch$_{ep300}$). (4) from the pretrained model for 120 epochs (TripLe$_{ep120}$). The results are given in Table 5.7.

When using 10% of the total training time (i.e., 300 epochs) for each trial, traditional multi-trial search only shows 0.221 Kendall-tau correlation. This indicates that correlation between the proxy accuracy (scratch$_{30}$) and the final training accuracy (scratch$_{300}$) is weak. On the other hand, TripLe shows a higher correlation to the final model performance trained under TripLe$_{120}$ and scratch$_{300}$.

We also evaluate the correlation between scratch$_{300}$ and TripLe$_{120}$, it shows a 0.363 Kendall-tau correction. This can be interpreted as the model that is suitable for scratch training may not fit for TripLe. Also, it can come from random seed selection [171] in the scratch training. For TripLe, the initialization weights are fixed.

Searched Model Comparison. We conduct 200 trials for both multi-trial NAS and NAS with TripLe (Learning curve in Appendix D). For each trial, we conduct 30 epochs of training using

Figure 5.4. Comparison of TripLe with knowledge distillation under 300 epochs of training for (a) Ti→S_{L24} and (b) Ti→S.

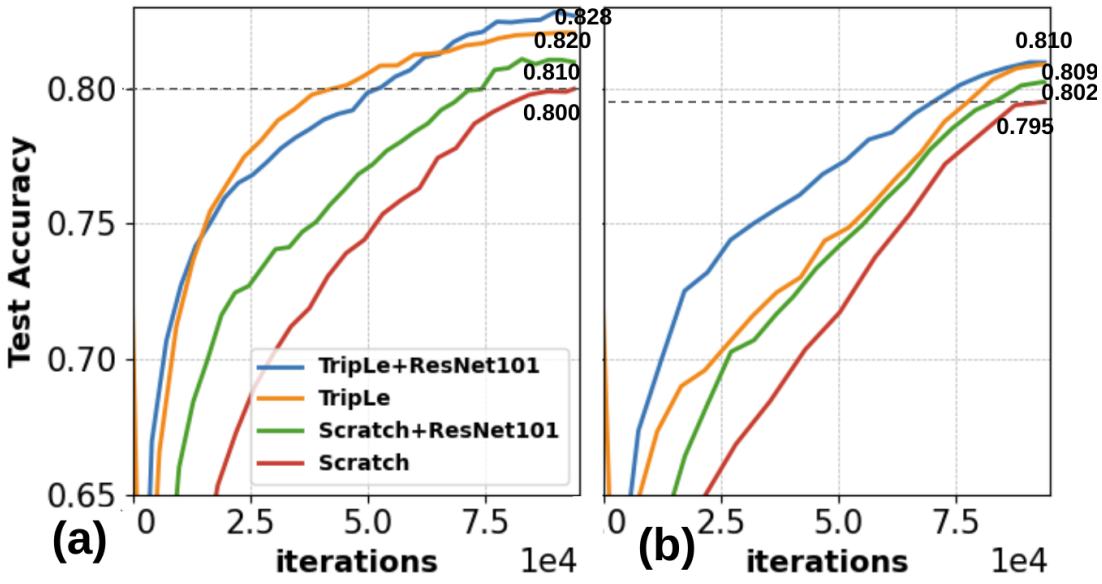


Table 5.8. Comparison results of using TripLe in multi-trial NAS and traditional multi-trial NAS.

	Search method	Evaluation method	Params (MB)	FLOPs (Million)	Test Acc (%)
DeiT-S (ours)	-	Scratch_{ep300}	22	8495	79.5
DeiT-B (ours)	-	Scratch_{ep300}	86	33722	81.0
ViT-scratch	Scratch_{ep30}	TripLe_{ep120}	30	11409	79.5
ViT-scratch	Scratch_{ep30}	TripLe_{ep300}	30	11409	80.8
ViT-TripLe	TripLe_{ep30}	TripLe_{ep120}	27	10416	79.7
ViT-TripLe	TripLe_{ep30}	TripLe_{ep300}	27	10416	81.1

TripLe (TripLe_{ep30}) and scratch training (Scratch_{ep30}). We set the FLOPs target ($FLOPs_0$) to 10000M and other hyperparameters for the regularized evolutionary and our reward function are given in Appendix A.2. As shown in Table 5.8, the model (ViT-TripLe) searched using NAS with TripLe can obtain 81.1% accuracy. ViT-TripLe outperforms our re-implement 86MB DeiT-B in task accuracy with 69%/69% reduction in parameter size and inference FLOPs. On the other hand, the model searched by traditional NAS can achieve 80.8% task accuracy (Detailed architectures in Appendix F).

5.6 Related Work

Reusing Pretrained Model and Progressive Learning. Methods that reuse pretrained models assume the small pretrained model pre-exists before starting the training. The smaller model will be scaled up to initialize the large model [21, 159]. For progressive learning, these methods assume the small pretrained model does not exist. The models are initialized randomly and grow towards the target model during training [50, 166, 139, 90, 40]. In this work, we assume the pretrained model exists before training and also employ progressive learning to grow the model during training. We compare both lines of work in Sec 5.5.

Efficient Transformer Learning. Besides, existing methods for the efficient transformer training techniques, such as pipeline parallelism [140, 67], large batch optimization [170], and layer dropping [175] are orthogonal to TripLe and works in reusing pretrained model. Some of the techniques are designed for NLP tasks, such as Electra [29] or token dropping [62], which cannot be directly applied in ViTs training. Knowledge distillation can also improve the quality and reduce training time [122, 151]. In this work, we compare and combine our approach with KD.

Neural Architecture Search. Recent advances in one-shot NAS leverage the idea of weight-sharing and train a super-network that contains all the possible model selections [164, 32, 94, 156, 49, 11]. For multi-trial NAS [185, 147, 148, 91], a controller samples candidate architectures and each one is trained from scratch. One shot is way faster than multi-trial method. However, one-shot cannot search training recipes and activation functions [32]. Also, one-shot incurs regularization conflict [49] that can hardly be resolved. In this work, we leverage TripLe to improve the performance of multi-trial NAS.

5.7 Conclusions

We propose TripLe, a method for scaling pretrained ViT to reduce the training time and improve task performance. Naïvely scaling the ViT once in multiple dimensions can hardly preserve the functionality of the pretrained model. Besides, the new parameters introduced

during scaling are under-trained and do not have their training states established. As such, TripLe scales the width of the model and optimizer states before training. During training, TripLe grows the depth by copying the warmed-up weights and optimizer states from existing layers. In this way, each expansion can mostly preserve functionality. The new weights added during depth expansion can also obtain their training states from the previous expansion stage.

In single-trial model scaling, TripLe not only reduces the training time of scaling ViTs but also achieves even better task accuracy compared to the baseline methods. In multi-trial NAS, the proxy accuracy obtained from TripLe shows a higher correlation to their final performance. Besides, the searched model with TripLe outperforms the counterpart obtained using traditional NAS in task accuracy.

Acknowledgement. Chapter 5, in part, contains a re-organized reprint of the material as it will appear in International Conference on Computer Vision (ICCV) 2023. Fu, Cheng; Hanxian Huang; Zixuan Jiang; Yun Ni; Lifeng Nai; Gang Wu; Liqun Cheng; Yanqi Zhou; Sheng Li; Andrew Li; Jishen Zhao. The dissertation author was the primary investigator and author of this paper.

Table 5.9. Hyperparameters for model scaling experiments. The hyperparameters are identical to DeiT-B. We find batch augmentation and Erasing are not useful to increase the final task accuracy.

Search method	Search method	Learning rate decay	Warmup epoch	Label smoothing	Dropout	Drop path	Repeat Aug	Gradient clip	RandAug [61]	Mixup [174]	Cutmix [172]	Erasing [179]
4096	4e-3	cosine	5	0.1	0.0	0.1	×	×	✓	✓	✓	✓

5.8 Appendix: Training Hyperparameters

5.8.1 Hyperparameters for Single-trial Model Scaling.

Our training hyperparameters are the same as DeiT-B [38] as given in Table 5.9. We find using repeat augmentation [12, 61] and erasing augmentation [179] doesn't show any performance improvement. As such, we do not use them in the training phase.

5.8.2 Hyperparameters for NAS

The regularized evolution algorithm discussed in Sec 5.4 is identical to AmoebaNet [126]. We set the population size set to 50 and the tournament size set to 10. The mutation probabilities are uniform and are identical to [126]. For the reward function, exponent $\varepsilon = -0.07$, FLOPs target is set to $FLOPs_0 = 10000M$.

5.9 Appendix: Details of Baseline Scaling Operators

5.9.1 Learning Curve of Scaling Operators

The learning curve for different expansion operators is given in Figure 5.6. The γ_{ST} with momentum information outperforms other baselines.

5.9.2 Details Explanations for bert2BERT and Learn-to-share

bert2BERT (γ_{b2B}): We use a simple example to illustrate the key idea of bert2BERT here. Assuming we are expanding the first layer w_0 and the input feature vector is d_{in}, the output of the matrix would be $d_o = d_{in}^T w_0$. w_0 has a dimension of 2×2 and d_{in} has a dimension of 2×1. After layer scaling, w_0'' has a size of 4×4.

$$d_{in} = \begin{bmatrix} a \\ b \end{bmatrix}, \qquad w_0 = \begin{bmatrix} o & p \\ q & r \end{bmatrix}, \qquad d_o^T = d_{in}^T w_0 = \begin{bmatrix} a_o \\ b_o \end{bmatrix} \tag{5.13}$$

When expanding the weight matrix w_0 given in Eq. 5.13 from a 2×2 matrix into a 4×4 matrix, we first expand the input dimensions.

We randomly select two rows, e.g., the first row, and duplicate them. Then, we normalize these rows based on the number of duplication. The corresponding input features will be duplicated in the same fashion without normalization. The result dense layers are given as follows:

$$d'_{in} = \begin{bmatrix} a \\ b \\ a \\ a \end{bmatrix}, \qquad w'_0 = \begin{bmatrix} \frac{o}{3} & \frac{p}{3} \\ q & r \\ \frac{o}{3} & \frac{p}{3} \\ \frac{o}{3} & \frac{p}{3} \end{bmatrix} \tag{5.14}$$

As is shown above, the result ${d'_{in}}^T w'_0 = d_{in}^T w_0$ does not change during the expansion. Next, we randomly select two columns, e.g., the second column, and duplicate them without normalization.

$$d''_{in} = \begin{bmatrix} a \\ b \\ a \\ a \end{bmatrix}, \qquad w''_0 = \begin{bmatrix} \frac{o}{3} & \frac{p}{3} & \frac{p}{3} & \frac{p}{3} \\ q & r & r & r \\ \frac{o}{3} & \frac{p}{3} & \frac{p}{3} & \frac{p}{3} \\ \frac{o}{3} & \frac{p}{3} & \frac{p}{3} & \frac{p}{3} \end{bmatrix} \tag{5.15}$$

The final output ($o'' = {d''_{in}}^T w''_0$) would be $o'' = [a_o, b_o, b_o, b_o]$. For the following layer w_1,

the input is determined and thus the policy of row duplication is determined as well. For w_1, we continue the same procedure for expanding columns (i.e., randomly select columns and duplicate them). And so on, the model functionality can be preserved.

$$LayerNorm(o) = \frac{(o'' - \mu_o)}{\sigma_o} \odot W^{LN} + b^{LN} \tag{5.16}$$

However, if the next layer is LayerNorm (Eq 5.16). The mean (μ_o) and variance (σ_o) of the output o changes. $\odot$ denotes the element-wise multiplication. During expansion, we don't know the relationship between a_o and b_o, so bert2BERT cannot preserve functionality through changing the LN scale and LN-bias, i.e. W^{LN} and b^{LN}.

On the other hand, γ_{ST} will yield output $o'' = [a_o, b_o, a_o, b_o]$. The mean μ_o and the variance σ_o of the output vector does not change.

Learn-to-grow. learn-to-grow [159] proposes to learn linear matrices that map the pretrained weights into larger weight matrices to preserve the functionality of the small pretrained model. We denote its width and depth expansion operator as γ_{ltg} and β_{ltg}, respectively.

$$W_i' = \gamma_{ltg}(w_i) = H_i w_i H_i^T, \qquad i \in \{1, ..., l\} \tag{5.17}$$

Here, H_i $(D \times d)$ is a trainable linear layer that maps the dense layer w_i into W_i'. w_i has a dimension of $d \times d$ and W_i' has a size of $D \times D$. For layer normalization and weight bias with a dimension of $d \times 1$, the expansion is similar to Eq 5.17 [159].

After width expansion, learn-to-share trains another set of linear mappings for depth expansion that expands W' into W:

$$W_i = \beta_{ltg}(w_i) = \Sigma_{j=1}^{l} P_{i,j} W_j', \qquad i \in \{1, ..., L\} \tag{5.18}$$

Here P_i is a $1 \times l$ vector. l is the number layers in the pretrained model; L is the number of layers in the scaled model. This means the expanded layer W_i is the weighted sum of W'_j where $j \in \{1, ..., l\}$,.

The linear mappings (H, P) are introduced to scale every dense layer in the scaled ViT. These mappings contain a large number of parameters and require a prohibitively expensive hardware memory for training. Some techniques are proposed in the paper to reduce the number of parameters, such as Kronecker factorization.

In this paper, we find the objective of training these linear mappings is the same as training the scaled model (Eq 5.6). For S→B, learn-to-grow can achieve 72% initial accuracy. Specifically, learn-to-grow trains the linear mapping H, P for around 200 steps and scale the model according to Eq 5.17-5.18. However, using γ_{ST} alone to scale S→B can achieve the pretrained DeiT-S accuracy (79%) at step 0. γ_{Pad0} can achieve 73% accuracy with 200 steps of model training. This means training these linear mappings to increase the initial accuracy is redundant. Besides, as discussed in Sec 5.2.3, we argue that the initial accuracy is not the key for successful model scaling.

5.10 Appendix: Combine TripLe with KD

As we reuse the DeiT architectures, the output has two parts: (1) the output logits of distillation head o_t and (2) the output logits of classification head o_s. Assuming the output logits of the teacher model is Z_t, the corresponding teaching label would be $y_t = \arg\max_c Z_t(c)$. When KD is applied, the hard loss is defined as Eq 5.19.

$$\mathscr{L}_{global}^{hardDistill} = \frac{1}{2}\mathscr{L}_{CE}(\psi(o_s), y) + \frac{1}{2}\mathscr{L}_{CE}(\psi(o_t), y_t) \tag{5.19}$$

ψ is the softmax function. $\mathscr{L}_{CE}$ is the cross-entropy loss. During model evaluation under KD, the prediction comes from the combination of both o_s and o_t: $\bar{y} = \arg\max_c \frac{o_s+o_t}{2}(c)$.

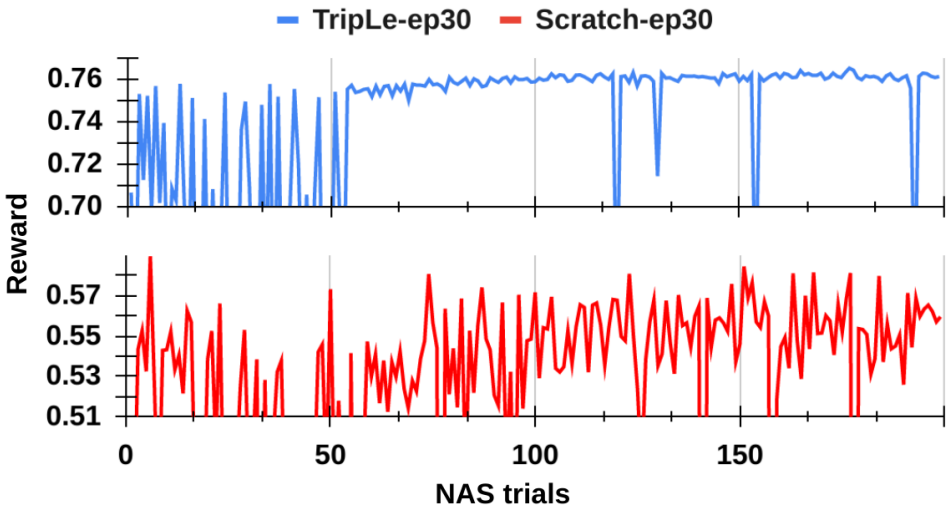


Figure 5.5. Learning Curve of the agents during NAS when each sample is trained with (1) TripLe$_{ep30}$ (2) Scratch$_{ep30}$.

When we disable the knowledge distillation, we follow the official DeiT implementation[2] for training and the loss is given as Eq 5.20.

$$\mathscr{L}_{global} = \frac{1}{2}\mathscr{L}_{CE}(\psi(\frac{o_s + o_t}{2}), y) \tag{5.20}$$

5.11 Appendix: Learning Curve of NAS

For each trial, both TripLe-NAS and multi-trial NAS conduct 30 epochs of training. The learning curve of the agent during the searching phase is given in Figure 5.5. Generally, both multi-trial and TripLe-NAS gradually increase reward over time. The learning curve of TripLe is more stable compared to multi-trial.

5.12 Appendix: Model Transfer Learning

Table 5.10 shows the transfer learning results of ViT-TripLe and ViT-Scratch. For the downstream tasks, the inputs are resized into 224×224.

[2]https://github.com/facebookresearch/deit

Table 5.10. Transfer learning results on various datasets.

Model	Params	FLOPs	CF-10	CF-100	Cars	Flowers
DeiT-B (official)	86M	33.7B	99.1	90.8	92.1	98.4
S→B, LTG	86M	33.7B	99.1	90.7	92.1	97.8
S→B, TripLe$_{ep300}$	86M	33.7B	99.1	90.8	92.2	98.4

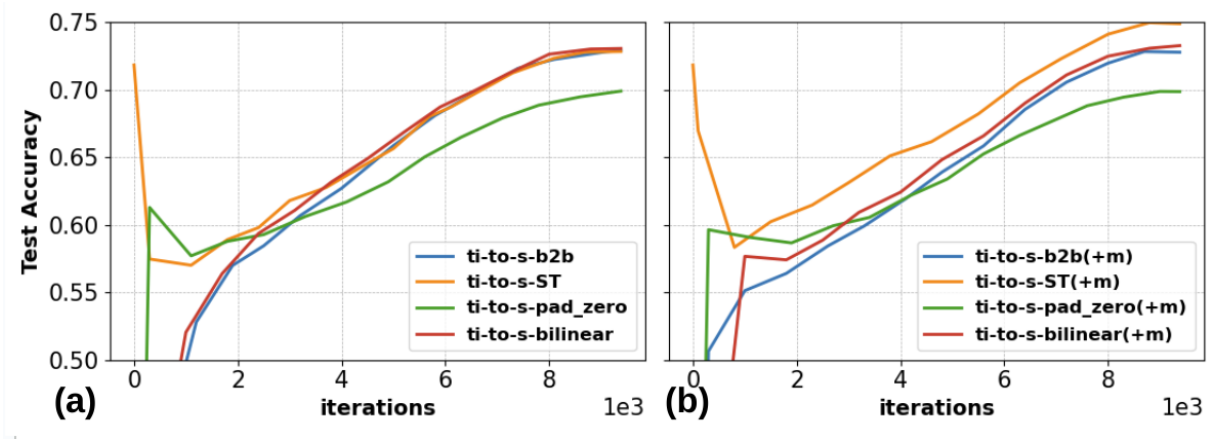


Figure 5.6. Training Ti→S with 30 epochs using different width expansion methods, i.e., γ_{b2B}, γ_{ST}, γ_{pad0}, γ_{intp}. '+m' denotes we also employ optimizer states in the pretrained model as discussed in Sec 5.2.3.

Table 5.11. Searched Architectures from (1) multi-trial NAS with TripLe and (2) traditional multi-trial NAS.

Model	Params	FLOPs	hidden dim	Layers	hf	ef	wd	lr
ViT-TripLe	27M	10416M	384	19	[32,32, 64,64,64,32,32,32,32,32,32,64] [32, 64,32,32,64,32,32]	[2,4,2,2,2,4,4,2,2,4,2,2] [4,4,2,2,2,4,2]	0.05	4e-3
ViT-Scratch	30M	11409M	384	19	[32,32,64,32,32,64,32,64,32,64,64,64] [32,32,32,32,32,32,32]	[3,4,4,2,3,4,2,3,4,4,4,2] [4,4,2,2,2,4,2]	0.05	4e-3

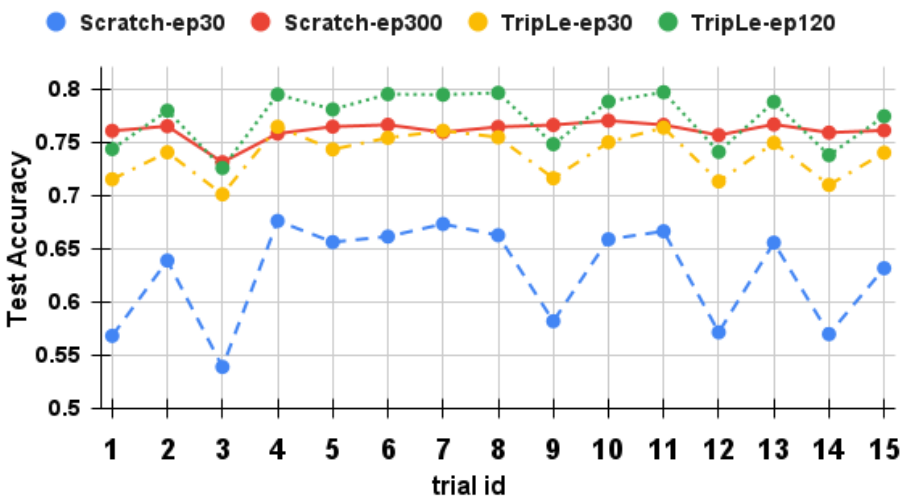


Figure 5.7. Task performance when trained with (1) TripLe$_{ep30}$(2) TripLe$_{ep120}$ (3) Scratch$_{ep30}$ (4) Scratch$_{ep30}$.

5.13 Appendix: Searched architectures.

Table 5.11 shows the models searched using NAS with TripLe and traditional multi-trial NAS.

Bibliography

[1] Gurobi optimization - the fastest solver. https://www.gurobi.com.

[2] oneapi deep neural network library (onednn). https://github.com/oneapi-src/oneDNN.

[3] Helib. Online: https://github.com/homenc/HElib, October 2021. EPFL-LDS.

[4] Samira Abnar, Mostafa Dehghani, and Willem Zuidema. Transferring inductive biases through knowledge distillation. *arXiv preprint arXiv:2006.00555*, 2020.

[5] John O Awoyemi, Adebayo O Adetunmbi, and Samuel A Oluwadare. Credit card fraud detection using machine learning techniques: A comparative analysis. In *2017 International Conference on Computing Networking and Informatics (ICCNI)*, pages 1–9. IEEE, 2017.

[6] Yu Bai, Yu-Xiang Wang, and Edo Liberty. Proxquant: Quantized neural networks via proximal operators. *arXiv preprint arXiv:1810.00861*, 2018.

[7] Jean-Claude Bajard, Julien Eynard, Anwar Hasan, and Vincent Zucca. A full rns variant of fv like somewhat homomorphic encryption schemes. Cryptology ePrint Archive, Report 2016/510, 2016. https://ia.cr/2016/510.

[8] Ankur Bapna, Naveen Arivazhagan, and Orhan Firat. Controlling computation versus quality for neural sequence models. *arXiv preprint arXiv:2002.07106*, 2020.

[9] Ankur Bapna and Orhan Firat. Simple, scalable adaptation for neural machine translation. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 1538–1548, Hong Kong, China, November 2019. Association for Computational Linguistics.

[10] Gabriel Bender, Pieter-Jan Kindermans, Barret Zoph, Vijay Vasudevan, and Quoc Le. Understanding and simplifying one-shot architecture search. In *International Conference on Machine Learning*, pages 549–558, 2018.

[11] Gabriel Bender, Hanxiao Liu, Bo Chen, Grace Chu, Shuyang Cheng, Pieter-Jan Kindermans, and Quoc V Le. Can weight sharing outperform random architecture search? an investigation with tunas. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 14323–14332, 2020.

[12] Maxim Berman, Hervé Jégou, Andrea Vedaldi, Iasonas Kokkinos, and Matthijs Douze. Multigrain: a unified image embedding for classes and instances. *arXiv preprint arXiv:1902.05509*, 2019.

[13] Joppe W Bos, Kristin Lauter, Jake Loftus, and Michael Naehrig. Improved security for a ring-based fully homomorphic encryption scheme. In *IMA International Conference on Cryptography and Coding*, pages 45–64. Springer, 2013.

[14] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. Fully homomorphic encryption without bootstrapping. Cryptology ePrint Archive, Report 2011/277, 2011. https://ia.cr/2011/277.

[15] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. *ACM Transactions on Computation Theory (TOCT)*, 6(3):1–36, 2014.

[16] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. 2020.

[17] Han Cai, Ligeng Zhu, and Song Han. Proxylessnas: Direct neural architecture search on target task and hardware. *arXiv preprint arXiv:1812.00332*, 2018.

[18] Zhaowei Cai, Xiaodong He, Jian Sun, and Nuno Vasconcelos. Deep learning with low precision by half-wave gaussian quantization. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 5918–5926, 2017.

[19] Rich Caruana. Multitask learning. *Machine learning*, 28(1):41–75, 1997.

[20] Bo Chang, Lili Meng, Eldad Haber, Frederick Tung, and David Begert. Multi-level residual networks from dynamical systems view. *arXiv preprint arXiv:1710.10348*, 2017.

[21] Cheng Chen, Yichun Yin, Lifeng Shang, Xin Jiang, Yujia Qin, Fengyu Wang, Zhi Wang, Xiao Chen, Zhiyuan Liu, and Qun Liu. bert2bert: Towards reusable pretrained language models. *arXiv preprint arXiv:2110.07143*, 2021.

[22] Chun-Fu Richard Chen, Quanfu Fan, and Rameswar Panda. Crossvit: Cross-attention multi-scale vision transformer for image classification. In *Proceedings of the IEEE/CVF international conference on computer vision*, pages 357–366, 2021.

[23] Tianqi Chen, Ian Goodfellow, and Jonathon Shlens. Net2net: Accelerating learning via knowledge transfer. *arXiv preprint arXiv:1511.05641*, 2015.

[24] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: An automated end-to-end optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 578–594, Carlsbad, CA, October 2018. USENIX Association.

[25] Wuyang Chen, Wei Huang, Xianzhi Du, Xiaodan Song, Zhangyang Wang, and Denny Zhou. Auto-scaling vision transformers without training. In *International Conference on Learning Representations*, 2022.

[26] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, et al. Dadiannao: A machine-learning supercomputer. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 609–622. IEEE, 2014.

[27] Edward Chou, Josh Beal, Daniel Levy, Serena Yeung, Albert Haque, and Li Fei-Fei. Faster cryptonets: Leveraging sparsity for real-world encrypted inference. *CoRR*, abs/1811.09953, 2018.

[28] Kevin Clark, Minh-Thang Luong, Urvashi Khandelwal, Christopher D. Manning, and Quoc V. Le. BAM! born-again multi-task networks for natural language understanding. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 5931–5937, Florence, Italy, July 2019. Association for Computational Linguistics.

[29] Kevin Clark, Minh-Thang Luong, Quoc V Le, and Christopher D Manning. Electra: Pre-training text encoders as discriminators rather than generators. *arXiv preprint arXiv:2003.10555*, 2020.

[30] CXL Consortium. Compute Express Link.

[31] Gen-Z Consortium. The Gen-Z Consortium.

[32] Xiaoliang Dai, Alvin Wan, Peizhao Zhang, Bichen Wu, Zijian He, Zhen Wei, Kan Chen, Yuandong Tian, Matthew Yu, Peter Vajda, et al. Fbnetv3: Joint architecture-recipe search using predictor pretraining. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 16276–16285, 2021.

[33] Zihang Dai, Zhilin Yang, Yiming Yang, Jaime Carbonell, Quoc Le, and Ruslan Salakhutdinov. Transformer-XL: Attentive language models beyond a fixed-length context. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 2978–2988, Florence, Italy, July 2019. Association for Computational Linguistics.

[34] Philip J Davis. *Interpolation and approximation*. Courier Corporation, 1975.

[35] Mostafa Dehghani, Josip Djolonga, Basil Mustafa, Piotr Padlewski, Jonathan Heek, Justin Gilmer, Andreas Steiner, Mathilde Caron, Robert Geirhos, Ibrahim Alabdulmohsin, et al. Scaling vision transformers to 22 billion parameters. *arXiv preprint arXiv:2302.05442*, 2023.

[36] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

[37] David Detlefs, Greg Nelson, and James B Saxe. Simplify: a theorem prover for program checking. *Journal of the ACM (JACM)*, 52(3):365–473, 2005.

[38] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics.

[39] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*, 2020.

[40] Utku Evci, Max Vladymyrov, Thomas Unterthiner, Bart van Merriënboer, and Fabian Pedregosa. Gradmax: Growing neural networks using gradient information. *arXiv preprint arXiv:2201.05125*, 2022.

[41] Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. *IACR Cryptol. ePrint Arch.*, 2012:144, 2012.

[42] Agner Fog et al. Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for intel, amd and via cpus. *Copenhagen University College of Engineering*, 93:110, 2011.

[43] Cheng Fu, Huili Chen, Zhenheng Yang, Farinaz Koushanfar, Yuandong Tian, and Jishen Zhao. Enhancing model parallelism in neural architecture search for multidevice system. *IEEE Micro*, 40(5):46–55, 2020.

[44] Cheng Fu, Shilin Zhu, Huili Chen, Farinaz Koushanfar, Hao Su, and Jishen Zhao. Simbnn: A similarity-aware binarized neural network acceleration framework. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 319–319. IEEE, 2019.

[45] Trevor Gale, Erich Elsen, and Sara Hooker. The state of sparsity in deep neural networks. *arXiv preprint arXiv:1902.09574*, 2019.

[46] Xinbo Gao, Bing Xiao, Dacheng Tao, and Xuelong Li. A survey of graph edit distance. *Pattern Analysis and applications*, 13(1):113–129, 2010.

[47] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the forty-first annual ACM symposium on Theory of computing*, pages 169–178, 2009.

[48] Ran Gilad-Bachrach, Nathan Dowlin, Kim Laine, Kristin Lauter, Michael Naehrig, and John Wernsing. Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy. In *International Conference on Machine Learning*, pages 201–210. PMLR, 2016.

[49] Chengyue Gong, Dilin Wang, Meng Li, Xinlei Chen, Zhicheng Yan, Yuandong Tian, Vikas Chandra, et al. Nasvit: Neural architecture search for efficient vision transformers with gradient conflict aware supernet training. In *International Conference on Learning Representations*, 2021.

[50] Linyuan Gong, Di He, Zhuohan Li, Tao Qin, Liwei Wang, and Tieyan Liu. Efficient training of bert by progressively stacking. In *International conference on machine learning*, pages 2337–2346. PMLR, 2019.

[51] Mitchell Gordon, Kevin Duh, and Nicholas Andrews. Compressing bert: Studying the effects of weight pruning on transfer learning. pages 143–155, 01 2020.

[52] Benjamin Graham, Alaaeldin El-Nouby, Hugo Touvron, Pierre Stock, Armand Joulin, Hervé Jégou, and Matthijs Douze. Levit: a vision transformer in convnet's clothing for faster inference. In *Proceedings of the IEEE/CVF international conference on computer vision*, pages 12259–12269, 2021.

[53] Demi Guo, Alexander M. Rush, and Yoon Kim. Parameter-efficient transfer learning with diff pruning, 2020.

[54] Fu-Ming Guo, Sijia Liu, Finlay S Mungall, Xue Lin, and Yanzhi Wang. Reweighted proximal pruning for large-scale language representation. *arXiv preprint arXiv:1909.12486*, 2019.

[55] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A Horowitz, and William J Dally. Eie: Efficient inference engine on compressed deep neural network. *ACM SIGARCH Computer Architecture News*, 44(3):243–254, 2016.

[56] Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015.

[57] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[58] Kartik Hegde, Jiyong Yu, Rohit Agrawal, Mengjia Yan, Michael Pellauer, and Christopher W. Fletcher. Ucnn: Exploiting computational reuse in deep neural networks via weight repetition. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ISCA '18, page 674–687. IEEE Press, 2018.

[59] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

[60] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, 2015.

[61] Elad Hoffer, Tal Ben-Nun, Itay Hubara, Niv Giladi, Torsten Hoefler, and Daniel Soudry. Augment your batch: Improving generalization through instance repetition. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 8129–8138, 2020.

[62] Le Hou, Richard Yuanzhe Pang, Tianyi Zhou, Yuexin Wu, Xinying Song, Xiaodan Song, and Denny Zhou. Token dropping for efficient bert pretraining. *arXiv preprint arXiv:2203.13240*, 2022.

[63] Neil Houlsby, Andrei Giurgiu, Stanislaw Jastrzebski, Bruna Morrone, Quentin De Laroussilhe, Andrea Gesmundo, Mona Attariyan, and Sylvain Gelly. Parameter-efficient transfer learning for nlp. In *International Conference on Machine Learning*, pages 2790–2799. PMLR, 2019.

[64] Jeremy Howard and Sebastian Ruder. Universal language model fine-tuning for text classification. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 328–339, Melbourne, Australia, July 2018. Association for Computational Linguistics.

[65] Hengyuan Hu, Rui Peng, Yu-Wing Tai, and Chi-Keung Tang. Network trimming: A data-driven neuron pruning approach towards efficient deep architectures. *CoRR*, abs/1607.03250, 2016.

[66] Junzhou Huang, Tong Zhang, and Dimitris Metaxas. Learning with structured sparsity. In *Proceedings of the 26th Annual International Conference on Machine Learning*, pages 417–424, 2009.

[67] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *Advances in neural information processing systems*, 32, 2019.

[68] Zhiheng Huang, Wei Xu, and Kai Yu. Bidirectional LSTM-CRF models for sequence tagging. *CoRR*, abs/1508.01991, 2015.

[69] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks. *Advances in neural information processing systems*, 29, 2016.

[70] Intel. Intel® Optane™ DC Persistent Memory, 2019.

[71] Eric Jang, Shixiang Gu, and Ben Poole. Categorical reparameterization with gumbel-softmax. *arXiv preprint arXiv:1611.01144*, 2016.

[72] Xiaoqi Jiao, Yichun Yin, Lifeng Shang, Xin Jiang, Xiao Chen, Linlin Li, Fang Wang, and Qun Liu. TinyBERT: Distilling BERT for natural language understanding. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 4163–4174, Online, November 2020. Association for Computational Linguistics.

[73] Mandar Joshi, Danqi Chen, Yinhan Liu, Daniel S Weld, Luke Zettlemoyer, and Omer Levy. Spanbert: Improving pre-training by representing and predicting spans. *Transactions of the Association for Computational Linguistics*, 8:64–77, 2020.

[74] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon. In-datacenter performance analysis of a tensor processing unit. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 1–12, June 2017.

[75] Sangil Jung, Changyong Son, Seohyung Lee, Jinwoo Son, Jae-Joon Han, Youngjun Kwak, Sung Ju Hwang, and Changkyu Choi. Learning to quantize deep networks by optimizing quantization intervals with task loss. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 4350–4359, 2019.

[76] Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan. {GAZELLE}: A low latency framework for secure neural network inference. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 1651–1669, 2018.

[77] Georgios A Kaissis, Marcus R Makowski, Daniel Rückert, and Rickmer F Braren. Secure, privacy-preserving and federated machine learning in medical imaging. *Nature Machine Intelligence*, 2(6):305–311, 2020.

[78] Maurice G Kendall. A new measure of rank correlation. *Biometrika*, 30(1/2):81–93, 1938.

[79] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[80] Jonathan Krause, Michael Stark, Jia Deng, and Li Fei-Fei. 3d object representations for fine-grained categorization. In *4th International IEEE Workshop on 3D Representation and Recognition (3dRR-13)*, Sydney, Australia, 2013.

[81] Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. 2009.

[82] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. Cifar-10 (canadian institute for advanced research).

[83] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. Cifar-100 (canadian institute for advanced research).

[84] imagenet Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[85] Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. Albert: A lite bert for self-supervised learning of language representations. *arXiv preprint arXiv:1909.11942*, 2019.

[86] Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. Albert: A lite bert for self-supervised learning of language representations. In *International Conference on Learning Representations*, 2020.

[87] Andrew Lavin and Scott Gray. Fast algorithms for convolutional neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4013–4021, 2016.

[88] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[89] Namhoon Lee, Thalaiyasingam Ajanthan, and Philip Torr. SNIP: SINGLE-SHOT NETWORK PRUNING BASED ON CONNECTION SENSITIVITY. In *International Conference on Learning Representations*, 2019.

[90] Changlin Li, Bohan Zhuang, Guangrun Wang, Xiaodan Liang, Xiaojun Chang, and Yi Yang. Automated progressive learning for efficient training of vision transformers. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 12486–12496, 2022.

[91] Sheng Li, Mingxing Tan, Ruoming Pang, Andrew Li, Liqun Cheng, Quoc V Le, and Norman P Jouppi. Searching for fast model families on datacenter accelerators. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 8085–8095, 2021.

[92] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. Microsoft coco: Common objects in context. In *European conference on computer vision*, pages 740–755. Springer, 2014.

[93] Chenxi Liu, Barret Zoph, Maxim Neumann, Jonathon Shlens, Wei Hua, Li-Jia Li, Li Fei-Fei, Alan Yuille, Jonathan Huang, and Kevin Murphy. Progressive neural architecture search. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 19–34, 2018.

[94] Hanxiao Liu, Karen Simonyan, and Yiming Yang. DARTS: Differentiable architecture search. In *International Conference on Learning Representations*, 2019.

[95] Hong Liu, Zhiyuan Li, David Hall, Percy Liang, and Tengyu Ma. Sophia: A scalable stochastic second-order optimizer for language model pre-training. *arXiv preprint arXiv:2305.14342*, 2023.

[96] Xiaodong Liu, Pengcheng He, Weizhu Chen, and Jianfeng Gao. Multi-task deep neural networks for natural language understanding. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 4487–4496, Florence, Italy, July 2019. Association for Computational Linguistics.

[97] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized BERT pretraining approach. *CoRR*, abs/1907.11692, 2019.

[98] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*, 2019.

[99] Zhuang Liu, Hanzi Mao, Chao-Yuan Wu, Christoph Feichtenhofer, Trevor Darrell, and Saining Xie. A convnet for the 2020s. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 11976–11986, 2022.

[100] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*, 2017.

[101] Qian Lou and Lei Jiang. She: A fast and accurate deep neural network for encrypted data. *Neural Information Processing Systems*, 2019.

[102] Christos Louizos, Max Welling, and Diederik P. Kingma. Learning sparse neural networks through l_0 regularization. In *International Conference on Learning Representations*, 2018.

[103] Christos Louizos, Max Welling, and Diederik P. Kingma. Learning sparse neural networks through l_0 regularization. In *International Conference on Learning Representations*, 2018.

[104] Jian-Hao Luo, Jianxin Wu, and Weiyao Lin. Thinet: A filter level pruning method for deep neural network compression. In *Proceedings of the IEEE international conference on computer vision*, pages 5058–5066, 2017.

[105] Azalia Mirhoseini, Hieu Pham, Quoc V Le, Benoit Steiner, Rasmus Larsen, Yuefeng Zhou, Naveen Kumar, Mohammad Norouzi, Samy Bengio, and Jeff Dean. Device placement optimization with reinforcement learning. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 2430–2439. JMLR. org, 2017.

[106] Adelabderrahmane Namani, Smail Kourta, Chris Cummins, Kim Hazelwood, Riyadh Baghdahdi, and Hugh Leather. Caviar: An e-graph based trs for automatic code optimization.

[107] Chandrakana Nandi, Max Willsey, Adam Anderson, James R. Wilcox, Eva Darulova, Dan Grossman, and Zachary Tatlock. Synthesizing structured cad models with equality saturation and inverse transformations. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 31–44, New York, NY, USA, 2020. Association for Computing Machinery.

[108] Charles Gregory Nelson. *Techniques for program verification*. Stanford University, 1980.

[109] Maria-Elena Nilsback and Andrew Zisserman. Automated flower classification over a large number of classes. In *2008 Sixth Indian Conference on Computer Vision, Graphics & Image Processing*, pages 722–729. IEEE, 2008.

[110] NVIDIA, Péter Vingelmann, and Frank H.P. Fitzek. Cuda, release: 10.2.89, 2020.

[111] Pavel Panchekha, Alex Sanchez-Stern, James R. Wilcox, and Zachary Tatlock. Automatically improving accuracy for floating point expressions. *SIGPLAN Not.*, 50(6):1–11, June 2015.

[112] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[113] Daiyi Peng, Xuanyi Dong, Esteban Real, Mingxing Tan, Yifeng Lu, Gabriel Bender, Hanxiao Liu, Adam Kraft, Chen Liang, and Quoc Le. Pyglove: Symbolic programming for automated machine learning. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 33, pages 96–108, 2020.

[114] Johan Perols. Financial statement fraud detection: An analysis of statistical and machine learning algorithms. *Auditing: A Journal of Practice & Theory*, 30(2):19–50, 2011.

[115] Jonas Pfeiffer, Andreas Rücklé, Clifton Poth, Aishwarya Kamath, Ivan Vulić, Sebastian Ruder, Kyunghyun Cho, and Iryna Gurevych. AdapterHub: A framework for adapting transformers. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 46–54, Online, October 2020. Association for Computational Linguistics.

[116] Hadi Pouransari, Zhucheng Tu, and Oncel Tuzel. Least squares binary quantization of neural networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops*, pages 698–699, 2020.

[117] Rohan Baskar Prabhakar, Sachit Kuhar, Rohit Agrawal, Christopher J. Hughes, and Christopher W. Fletcher. Summerge: An efficient algorithm and implementation for weight repetition-aware dnn inference. In *Proceedings of the ACM International Conference on Supercomputing*, ICS '21, page 279–290, New York, NY, USA, 2021. Association for Computing Machinery.

[118] Varot Premtoon, James Koppel, and Armando Solar-Lezama. Semantic code search via equational reasoning. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 1066–1082, New York, NY, USA, 2020. Association for Computing Machinery.

[119] W Nicholson Price and I Glenn Cohen. Privacy in the age of medical big data. *Nature medicine*, 25(1):37–43, 2019.

[120] Pytorch-Sparse. Pytorch Sparse Library.

[121] Charles R Qi, Hao Su, Kaichun Mo, and Leonidas J Guibas. Pointnet: Deep learning on point sets for 3d classification and segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 652–660, 2017.

[122] Yujia Qin, Yankai Lin, Jing Yi, Jiajie Zhang, Xu Han, Zhengyan Zhang, Yusheng Su, Zhiyuan Liu, Peng Li, Maosong Sun, et al. Knowledge inheritance for pre-trained language models. *arXiv preprint arXiv:2105.13880*, 2021.

[123] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *SIGPLAN Not.*, 48(6):519–530, June 2013.

[124] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. In *European conference on computer vision*, pages 525–542. Springer, 2016.

[125] Elad Ben-Zaken1 Shauli Ravfogel and Yoav Goldberg. Bitfit: Simple parameter-efficient fine-tuning for transformer-based masked language-models.

[126] Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V Le. Regularized evolution for image classifier architecture search. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 4780–4789, 2019.

[127] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 779–788, 2016.

[128] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. *Advances in neural information processing systems*, 28:91–99, 2015.

[129] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. In *International Conference on Medical image computing and computer-assisted intervention*, pages 234–241. Springer, 2015.

[130] Amit Sabne. Xla : Compiling machine learning for peak performance, 2020.

[131] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4510–4520, 2018.

[132] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. Distilbert, a distilled version of BERT: smaller, faster, cheaper and lighter. *CoRR*, abs/1910.01108, 2019.

[133] Alexander Schrijver. *Theory of linear and integer programming*. John Wiley & Sons, 1998.

[134] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[135] Microsoft SEAL (release 3.0). http://sealcrypto.org, October 2018. Microsoft Research, Redmond, WA.

[136] Ali Shafiee, Anirban Nag, Naveen Muralimanohar, Rajeev Balasubramonian, John Paul Strachan, Miao Hu, R Stanley Williams, and Vivek Srikumar. Isaac: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars. *ACM SIGARCH Computer Architecture News*, 44(3):14–26, 2016.

[137] Hardik Sharma, Jongse Park, Naveen Suda, Liangzhen Lai, Benson Chau, Vikas Chandra, and Hadi Esmaeilzadeh. Bit fusion: Bit-level dynamically composable architecture for accelerating deep neural network. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 764–775. IEEE, 2018.

[138] Sheng Shen, Zhen Dong, Jiayu Ye, Linjian Ma, Zhewei Yao, Amir Gholami, Michael W Mahoney, and Kurt Keutzer. Q-bert: Hessian based ultra low precision quantization of bert. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 8815–8821, 2020.

[139] Sheng Shen, Pete Walsh, Kurt Keutzer, Jesse Dodge, Matthew Peters, and Iz Beltagy. Staged training for transformer language models. In *International Conference on Machine Learning*, pages 19893–19908. PMLR, 2022.

[140] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

[141] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *ICLR*, 2015.

[142] Michael Stepp, Ross Tate, and Sorin Lerner. Equality-based translation validator for llvm. In *International Conference on Computer Aided Verification*, pages 737–742. Springer, 2011.

[143] Asa Cooper Stickland and Iain Murray. Bert and pals: Projected attention layers for efficient adaptation in multi-task learning. In *International Conference on Machine Learning*, pages 5986–5995. PMLR, 2019.

[144] Mengshu Sun, Zhengang Li, Alec Lu, Yanyu Li, Sung-En Chang, Xiaolong Ma, Xue Lin, and Zhenman Fang. Film-qnn: Efficient fpga acceleration of deep neural networks with intra-layer, mixed-precision quantization. In *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '22, page 134–145, New York, NY, USA, 2022. Association for Computing Machinery.

[145] Zhiqing Sun, Hongkun Yu, Xiaodan Song, Renjie Liu, Yiming Yang, and Denny Zhou. MobileBERT: a compact task-agnostic BERT for resource-limited devices. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 2158–2170, Online, July 2020. Association for Computational Linguistics.

[146] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S Emer. Efficient processing of deep neural networks: A tutorial and survey. *Proceedings of the IEEE*, 105(12):2295–2329, 2017.

[147] Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, Mark Sandler, Andrew Howard, and Quoc V Le. Mnasnet: Platform-aware neural architecture search for mobile. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 2820–2828, 2019.

[148] Mingxing Tan and Quoc Le. Efficientnet: Rethinking model scaling for convolutional neural networks. In *International Conference on Machine Learning*, pages 6105–6114. PMLR, 2019.

[149] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. Equality saturation: a new approach to optimization. In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 264–276, 2009.

[150] Tensorflow-Sparse. Tensorflow sparse matrix API.

[151] Hugo Touvron, Matthieu Cord, Matthijs Douze, Francisco Massa, Alexandre Sablayrolles, and Hervé Jégou. Training data-efficient image transformers & distillation through attention. In *International conference on machine learning*, pages 10347–10357. PMLR, 2021.

[152] Hugo Touvron, Matthieu Cord, and Hervé Jégou. Deit iii: Revenge of the vit. In *Computer Vision–ECCV 2022: 17th European Conference, Tel Aviv, Israel, October 23–27, 2022, Proceedings, Part XXIV*, pages 516–533. Springer, 2022.

[153] Hugo Touvron, Matthieu Cord, Alexandre Sablayrolles, Gabriel Synnaeve, and Hervé Jégou. Going deeper with image transformers. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 32–42, 2021.

[154] Ahmet Üstün, Arianna Bisazza, Gosse Bouma, and Gertjan van Noord. UDapter: Language adaptation for truly Universal Dependency parsing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 2302–2315, Online, November 2020. Association for Computational Linguistics.

[155] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

[156] Alvin Wan, Xiaoliang Dai, Peizhao Zhang, Zijian He, Yuandong Tian, Saining Xie, Bichen Wu, Matthew Yu, Tao Xu, Kan Chen, Peter Vajda, and Joseph E. Gonzalez. Fbnetv2: Differentiable neural architecture search for spatial and channel dimensions. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2020.

[157] Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel Bowman. GLUE: A multi-task benchmark and analysis platform for natural language understanding. In *Proceedings of the 2018 EMNLP Workshop BlackboxNLP: Analyzing and Interpreting Neural Networks for NLP*, pages 353–355, Brussels, Belgium, November 2018. Association for Computational Linguistics.

[158] Kuan Wang, Zhijian Liu, Yujun Lin, Ji Lin, and Song Han. Haq: Hardware-aware automated quantization with mixed precision. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2019.

[159] Peihao Wang, Rameswar Panda, Lucas Torroba Hennigen, Philip Greengard, Leonid Karlinsky, Rogerio Feris, David Daniel Cox, Zhangyang Wang, and Yoon Kim. Learning to grow pretrained models for efficient transformer training. In *International Conference on Learning Representations*, 2023.

[160] Yisu Remy Wang, Shana Hutchison, Jonathan Leang, Bill Howe, and Dan Suciu. Spores: Sum-product optimization via relational equality saturation for large scale linear algebra. *Proc. VLDB Endow.*, 13(12):1919–1932, July 2020.

[161] Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256, 1992.

[162] Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. Egg: Fast and extensible equality saturation. *Proc. ACM Program. Lang.*, 5(POPL), January 2021.

[163] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam

Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. Huggingface's transformers: State-of-the-art natural language processing, 2020.

[164] Bichen Wu, Xiaoliang Dai, Peizhao Zhang, Yanghan Wang, Fei Sun, Yiming Wu, Yuandong Tian, Peter Vajda, Yangqing Jia, and Kurt Keutzer. Fbnet: Hardware-aware efficient convnet design via differentiable neural architecture search. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 10734–10742, 2019.

[165] Ji Xin, Raphael Tang, Jaejun Lee, Yaoliang Yu, and Jimmy Lin. Deebert: Dynamic early exiting for accelerating bert inference. *arXiv preprint arXiv:2004.12993*, 2020.

[166] Cheng Yang, Shengnan Wang, Chao Yang, Yuechuan Li, Ru He, and Jingqiao Zhang. Progressively stacking 2.0: A multi-stage layerwise training method for bert training speedup. *arXiv preprint arXiv:2011.13635*, 2020.

[167] Yichen Yang, Phitchaya Phothilimthana, Yisu Wang, Max Willsey, Sudip Roy, and Jacques Pienaar. Equality saturation for tensor graph superoptimization. *Proceedings of Machine Learning and Systems*, 3, 2021.

[168] Zhilin Yang, Zihang Dai, Yiming Yang, Jaime Carbonell, Ruslan Salakhutdinov, and Quoc V Le. Xlnet: Generalized autoregressive pretraining for language understanding. *arXiv preprint arXiv:1906.08237*, 2019.

[169] Serena Yeung, Francesca Rinaldo, Jeffrey Jopling, Bingbin Liu, Rishab Mehra, N Lance Downing, Michelle Guo, Gabriel M Bianconi, Alexandre Alahi, Julia Lee, et al. A computer vision system for deep learning-based detection of patient mobilization activities in the icu. *NPJ digital medicine*, 2(1):11, 2019.

[170] Yang You, Jing Li, Sashank Reddi, Jonathan Hseu, Sanjiv Kumar, Srinadh Bhojanapalli, Xiaodan Song, James Demmel, Kurt Keutzer, and Cho-Jui Hsieh. Large batch optimization for deep learning: Training bert in 76 minutes. *arXiv preprint arXiv:1904.00962*, 2019.

[171] Kaicheng Yu, René Ranftl, and Mathieu Salzmann. An analysis of super-net heuristics in weight-sharing nas. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 44(11):8110–8124, 2021.

[172] Sangdoo Yun, Dongyoon Han, Seong Joon Oh, Sanghyuk Chun, Junsuk Choe, and Youngjoon Yoo. Cutmix: Regularization strategy to train strong classifiers with localizable features. In *Proceedings of the IEEE/CVF international conference on computer vision*, pages 6023–6032, 2019.

[173] Dongqing Zhang, Jiaolong Yang, Dongqiangzi Ye, and Gang Hua. Lq-nets: Learned quantization for highly accurate and compact deep neural networks. In *Proceedings of the European conference on computer vision (ECCV)*, pages 365–382, 2018.

[174] Hongyi Zhang, Moustapha Cisse, Yann N Dauphin, and David Lopez-Paz. mixup: Beyond empirical risk minimization. *arXiv preprint arXiv:1710.09412*, 2017.

[175] Minjia Zhang and Yuxiong He. Accelerating training of transformer-based language models with progressive layer dropping. *Advances in Neural Information Processing Systems*, 33:14011–14023, 2020.

[176] Wei Zhang, Lu Hou, Yichun Yin, Lifeng Shang, Xiao Chen, Xin Jiang, and Qun Liu. TernaryBERT: Distillation-aware ultra-low bit BERT. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 509–521, Online, November 2020. Association for Computational Linguistics.

[177] Wentai Zhang, Hanxian Huang, Jiaxi Zhang, Ming Jiang, and Guojie Luo. Adaptive-precision framework for sgd using deep q-learning. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8. IEEE, 2018.

[178] Xiangyu Zhang, Xinyu Zhou, Mengxiao Lin, and Jian Sun. Shufflenet: An extremely efficient convolutional neural network for mobile devices. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 6848–6856, 2018.

[179] Zhun Zhong, Liang Zheng, Guoliang Kang, Shaozi Li, and Yi Yang. Random erasing data augmentation. In *Proceedings of the AAAI conference on artificial intelligence*, volume 34, pages 13001–13008, 2020.

[180] Aojun Zhou, Anbang Yao, Yiwen Guo, Lin Xu, and Yurong Chen. Incremental network quantization: Towards lossless cnns with low-precision weights. *arXiv preprint arXiv:1702.03044*, 2017.

[181] Daquan Zhou, Bingyi Kang, Xiaojie Jin, Linjie Yang, Xiaochen Lian, Zihang Jiang, Qibin Hou, and Jiashi Feng. Deepvit: Towards deeper vision transformer. *arXiv preprint arXiv:2103.11886*, 2021.

[182] Shuchang Zhou, Yuxin Wu, Zekun Ni, Xinyu Zhou, He Wen, and Yuheng Zou. Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients. *CoRR*, abs/1606.06160, 2016.

[183] Yanqi Zhou, Xuanyi Dong, Tianjian Meng, Mingxing Tan, Berkin Akin, Daiyi Peng, Amir Yazdanbakhsh, Da Huang, Ravi Narayanaswami, and James Laudon. Towards the co-design of neural networks and accelerators. *Proceedings of Machine Learning and Systems*, 4:141–152, 2022.

[184] Chenzhuo Zhu, Song Han, Huizi Mao, and William J Dally. Trained ternary quantization. *arXiv preprint arXiv:1612.01064*, 2016.

[185] Barret Zoph and Quoc V Le. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578*, 2016.

[186] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V Le. Learning transferable architectures for scalable image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 8697–8710, 2018.