计 算 机 研 究 与 发 展
Journal of Computer Research and Development
https://crad.ict.ac.cn
62(12): 3122−3146, 2025

一种可动态伸缩的移动端深度计算图算优化方法

罗诗妍[1] 刘思聪[1] 郭 斌[1] 方 程[1] 王敏帆[1] 郭 赛[1] 於志文[1,2]
[1](西北工业大学计算机学院 西安 710072)
[2](哈尔滨工程大学 哈尔滨 150006)
(luosy0107@mail.nwpu.edu.cn)

Graph Operator-Based Dynamically Scalable Optimization Method for Deep Computing on Mobile Devices

Luo Shiyan[1], Liu Sicong[1], Guo Bin[1], Fang Cheng[1], Wang Minfan[1], Guo Sai[1], and Yu Zhiwen[1,2]
[1](*School of Computer Science*, *Northwestern Polytechnical University*, *Xi'an* 710072)
[2](*Harbin Engineering University*, *Harbin* 150006)

Abstract In recent years, it has become a trend to introduce deep neural network (DNN) into mobile devices. Many applications that facilitate daily life, such as voice assistants and activity recognition, have been integrated into smartphones, wearable devices, and embedded systems. However, it is challenging to deploy CPU-bound DNN on mobile devices with limited resources, such as computing power, storage, and battery. Existing methods, such as manually designed DNN compression techniques and automated on-demand DNN compression techniques, are limited to optimizing model structures. It restricts the upper limit of performance optimization for DNN deployment and makes it difficult to adapt to devices with extremely constrained resources. In addition, these statically pre-designed optimization methods do not consider the resource contention and dynamic demand characteristics of the deployment environment in mobile applications. The inability to adjust strategies in real-time under dynamic environment results in suboptimal accuracy-efficiency performance. To address these challenges, we propose AdaInfer, a runtime-scalable cross-layer optimization method for DNN. AdaInfer adaptively selects the optimal comprehensive deployment strategy for model layers, computational graph layers, and memory layers based on current hardware resource constraints and user performance requirements to optimize multiple performance metrics. It also adjusts the optimal strategy in real-time as the scenario changes. Specifically, we design a scalable graph-computation structure that is model-agnostic and a corresponding cross-layer optimization strategy. These are capable of automatically adjusting to maximize deployment efficiency on heterogeneous devices. Then, we model the runtime adjustment problem of the algorithm-system cross-layer optimization strategy as a dynamic optimization problem and represent the dynamic environment through a set of runtime-varying resource constraints. We also propose an efficient search strategy to enhance the efficiency and quality of local online searches. In evaluations conducted on three types of mobile and edge devices, five models, and four continuously changing mobile scenarios, experimental results show that compared with previous work, AdaInfer reduces memory usage by up to 42.35% and latency by up to 73.89% without significantly affecting accuracy.

收稿日期: 2024-08-29; **修回日期**: 2025-04-11
基金项目: 国家杰出青年科学基金项目(62025205); 国家自然科学基金项目(62032020, 62102317, 62532009, 62522215)
This work was supported by the National Science Foundation of China for Distinguished Young Scholars (62025205) and the National Natural Science Foundation of China (62032020, 62102317, 62532009, 62522215).
通信作者: 刘思聪(scliu@nwpu.edu.cn)

Key words deep learning; mobile computing; model compression; computation graph optimization; memory scheduling

摘 要 近年来，将深度神经网络（deep neural network, DNN）引入移动设备成为一种趋势. 智能手机、可穿戴设备和嵌入式设备上集成了许多便利生活的应用，如语音助手和活动识别. 然而，在资源受限（如算力、存储和电池）移动终端部署计算密集型深度模型具有挑战. 现有方法如手工设计的 DNN 压缩技术和自动化按需 DNN 压缩技术局限于优化模型结构，限制了 DNN 部署的性能优化上限，难以适应资源极度受限的终端设备. 此外，已有静态预设计的优化方法未考虑移动应用部署环境的资源争用和动态需求特性，在动态环境下无法及时调整策略，从而导致次优精度-效率表现. 为了解决这些挑战，提出了 AdaInfer，一种在运行时可伸缩的 DNN 跨层优化方法. AdaInfer 根据当前硬件资源限制及用户性能需求，自适应选择模型层、计算图层和内存层的最佳综合部署策略以优化多个性能指标，并随着场景变化及时调整最优策略. 具体而言，设计了一种模型无关的可伸缩图算结构和对应的跨层优化策略，能够在异构设备上自动调整以最大化部署效率. 随后，将算法-系统跨层优化策略的运行时调整问题建模为动态优化问题，并通过一组运行时变化的资源约束来建模动态环境. 还提出了一种高效搜索策略，以提高本地在线搜索效率和质量. 实验结果显示，在 3 种典型移动和边缘设备、5 种模型和 4 种持续变化移动场景的评估中，AdaInfer 与先前的工作相比，在不显著影响精度的前提下，将内存占用最多降低了 42.35%，时延最多降低了 73.89%.

关键词 深度学习；移动计算；模型压缩；计算图优化；内存调度

中图法分类号 TP391

DOI: 10.7544/issn1000-1239.202440701 **CSTR:** 32373.14.issn1000-1239.202440701

近年来，智能手机、可穿戴设备和嵌入式设备等人工智能物联网（artificial intelligence of things, AIoT）终端不断集成持续本地运行的应用程序，极大地便利了人们的生活[1]. 例如，智能手机语音助手（如 Proxitalk[2]）和可穿戴传感器支持的活动识别（如 IMUTube[3]）. 将深度神经网络（deep neural network, DNN）驱动的计算技术引入移动设备已成为趋势，显著提升了终端设备的数据分析能力. 此外，由于用户对数据传输成本和隐私的关注日益增加，在终端本地执行 DNN 成为理想的移动计算范式[4]. 然而，DNN 通常具有较高的计算和存储复杂度，对内存、算力、电量资源需求较高. 而移动终端设备资源受限、资源供需悬殊加剧了深度模型轻量化技术在保持原精度和满足实时性、低功耗间的矛盾. 特别是在低成本、资源极端受限的终端设备（如 MCU, DSP）上运行模型时，这一矛盾更为显著.

现有轻量化深度学习研究在算法和底层加速方面进行了大量探索，以平衡应用性能（如准确性、延迟）和资源约束（如电池、存储）之间的矛盾. 主要技术包括：1）手工设计的 DNN 优化方法（如权重剪枝[5]、低秩分解[6]、并行化[7]），通过人工设计降低模型复杂度；2）自动化的按需 DNN 优化方法（如 AMC[8]、AdaDeep[9]、神经架构搜索[10]）使用元学习器或搜索算法为不同设备平台自动找到最优的 DNN 优化策略. 然而，由于移动端设备具有多维动态情境和多层耦合系统的特点，这些方法在实际部署中存在模型计算效率较低的问题.

在实际应用中，移动终端的 DNN 部署环境往往高度动态且难以预测. 这些动态变化使得资源受限条件下深度计算的精度和实时性能需求受环境变化的影响. 现有方法基于静态情境预先搜索得到的最佳模型结构和部署方案，在运行时由于设备情境变化难以持续保持最优计算效率. 因此，需要研究如何在设备运行时动态调整模型优化方案，以实现情境最优适配的可伸缩计算. 具体来说，多维动态情境包括时间变化的硬件能力（如存储、电池、处理器）、DNN 的活跃执行时间、实际环境触发的推理频率，以及其他应用程序引起的资源争用. 如图 1 所示，自动驾驶汽车车载计算设备的语义分割 DNN 计算任务（如 Dabnet[11]）需持续感知环境输入数据流. 在行驶过程中，设备的电池因 DNN 执行、内存访问和传感器采样的不可预测频率而动态消耗，形成了 DNN 部署的动态资源约束. 同时，存储单元和计算核也可能被其他车载应用动态占用，导致 DNN 参数的存储预算变化.

面对移动终端动态情境，深度计算系统部署优

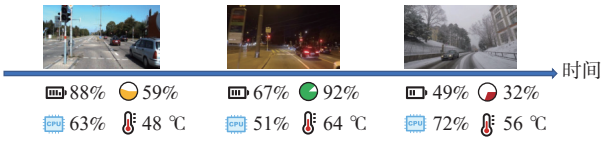


Fig. 1 Changes of scenarios and resources in autonomous driving applications

图 1 自动驾驶应用中场景与资源变化

化方案需要在模型层、计算图层和内存调度层具备高度可伸缩性,以在特定设备上实现最佳部署效率. 然而,移动终端嵌入式智能计算系统是一个多层耦合系统,其整体计算效率不仅受到模型层结构影响,还受到计算图层拓扑和内存调度层策略的影响. 现有方法通常基于单层空间设计优化,对具有不同组成单元和拓扑结构的跨层空间、终端多层计算系统的优化效果十分有限[12]. 因此,需要研究如何联合优化资源适配的深度模型和模型自适应的底层存算资源调度(即算法-系统跨层技术),涉及模型、计算图、内存调度等多个层面要素. 然而,为了实现这一目标,仍然面临 2 个方面的挑战:

1)设计与模型结构无关的弹性图算结构及跨层优化方法. 由于 DNN 模型种类繁多且具有不同的存算特性,设计通用的可伸缩图算结构需兼容多种模型单元,并协调模型层、计算图层与内存调度层优化空间的冲突部分. 此外,跨层方法需要协调层间数据流依赖以避免不一致问题,并协同设计优化方法以实现可叠加的性能提升.

2)运行时基于多维动态情境自动搜索和匹配最优跨层优化方案. 在动态情境下,需要快速搜索最优跨层方案以适应波动的可用资源和优化目标. 然而,涵盖模型层、计算图层和内存调度层的跨层优化方案存在庞大的搜索空间,需要在广泛的组合中快速评估并搜索最优策略,同时平衡计算效率、内存利用率和延迟等多个性能指标.

针对这些挑战和限制,我们提出了 AdaInfer,一种在运行时根据动态情境的 DNN 跨层优化方法,如图 2 所示. AdaInfer 通过快速调整和组合模型层、计算图层和内存层的优化策略,持续平衡多个性能指标. 我们设计了一种模型结构无关的可伸缩图算结构,并基于该结构提出了基于模型层(见 3.1 节)、计算图层(见 3.2 节)和内存调度层(见 3.3 节)的自适应跨层优化方法,能够根据不同设备的硬件资源和平台自动调整以最大化部署效率. 随后,我们将跨层优化策略的运行时调整问题建模为一个多目标优化问题(见式(4)),通过一组运行时变化的资源约束(如内存、时延和准确率)来建模动态环境. 我们提出了一种高效搜索策略,包括统一跨层精简搜索空间的构建(见 4.1 节)、基于进化算法的跨层优化策略搜索(见 4.2 节)和运行时在线性能评估(见 4.3 节),以提高本地在线搜索效率和质量.

本文的主要贡献如下:

1)提出了首个运行时自动调整的 DNN 跨层优化方法 AdaInfer,基于弹性图算结构和跨层协同优化方法,在运行时联合搜索最佳跨层优化方案,实现了 DNN 在异构设备上的高效部署,并在资源波动情况下自适应满足动态优化目标.

2)AdaInfer 提出了一种与模型结构无关的弹性图算结构,结合可替换压缩模块、计算图融合与算子重排,实现了模型、计算图和内存调度的跨层优化.

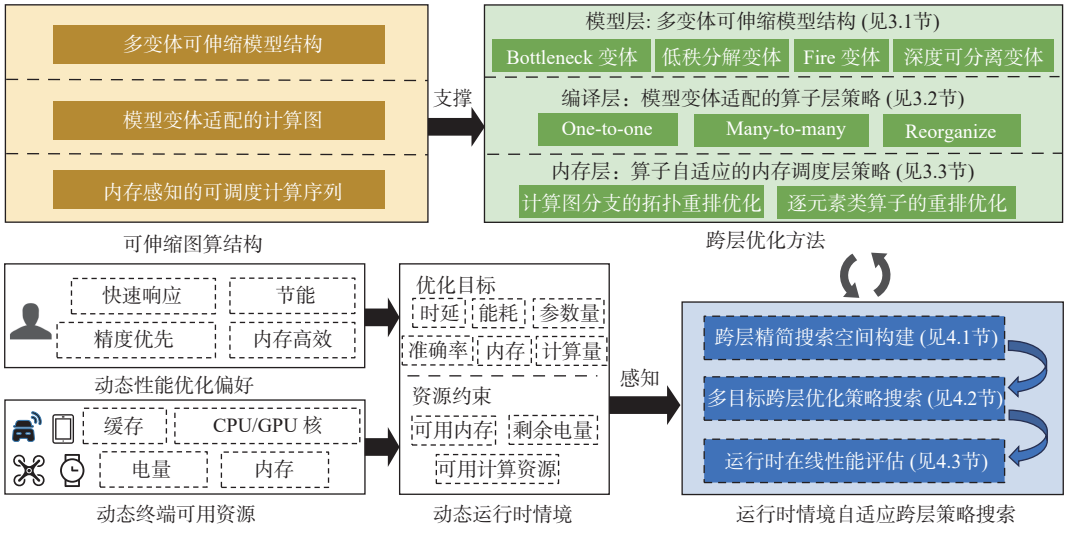


Fig. 2 Framework diagram of AdaInfer method

图 2 AdaInfer 方法框架图

此外，通过精简搜索空间和基于进化算法的优化策略搜索，结合高斯过程回归加速评估，AdaInfer能够在设备运行时在线搜索和调整跨层优化方案.

3）通过在3种常见设备（Jetson、树莓派和小米9手机）上测试不同移动应用程序的5种模型，以及在多种持续变化的移动场景中自适应搜索部署方案，实验结果展示了AdaInfer在运行时情境动态变化中的优势. AdaInfer能够根据当前硬件资源限制及用户性能需求，自适应选择最佳部署策略，并随着场景变化及时调整最优方案. 与先前的工作相比，AdaInfer在不显著影响精度的前提下，将内存占用最多降低了42.35%，时延最多降低了73.89%.

1 相关工作

随着物联网泛在移动/嵌入式设备的普及和能力的提升，深度计算逐渐向终端下沉成为前沿趋势[13-15]. 智能物联网的资源异构性、动态性、响应实时性等为自适应人工智能带来了新的挑战，例如资源受限移动终端深度模型轻量化技术、算子融合技术以及内存优化技术等方面的研究目前已在国内外多个领域获得重视并取得系列研究成果. 本节将对这些技术进行介绍.

1.1 深度模型轻量化技术

模型压缩技术旨在通过简化模型复杂性、减少参数量和优化结构来缩小模型规模，以适应资源受限的移动终端设备. 典型的模型压缩技术包括参数剪枝、量化、低秩分解和模型蒸馏. 参数剪枝通过移除模型中的一些参数（如神经元或权重）来减小模型. He等人[16]提出了一种新的通道剪枝方法，通过基于LASSO回归的通道选择和最小二乘重建，有效地对训练好的模型每一层进行剪枝. Liberis等人[17]提出了一种用于卷积神经网络的可微结构化剪枝方法，该方法与以往的网络剪枝工作相比，更好地利用了MCU的有限资源，并依赖于参数重要性反馈来获得压缩程度高但精度依然准确的模型. 量化是将模型中的浮点数值转换为较低精度的表示（例如，将32 b浮点数转换为8 b整数）. Zhou等人[18]提出了深度模型量化的优化框架. 该框架先用一种方法来估计各个层中参数量化误差对整体模型预测精度的影响程度；然后基于测量方法的优化过程找到每层的最佳量化位宽. 为了充分探索增量模型的相似性并且提高压缩率，Jin等人[19]通过开发基于层的局部敏感量化方案和误差反馈机制，提出了一种基于量化的局部敏感增量压缩方法. 该方法在训练过程中动态纠正量化误差以避免量化误差降低恢复模型的性能. 低秩分解通过分解模型中的权重矩阵来减小模型. Sainath等人[20]发现DNN为了获得良好的性能通过使用大量参数进行训练导致训练速度非常慢，这些参数大部分都在最终权重层中，因此提出了最终权重层的低秩矩阵分解，将网络参数量减少了30%~50%的同时减少训练时间并保持精度. Cai等人[21]首先将损失函数和压缩成本函数组合成一个联合函数，并将其作为优化框架进行优化；然后将CUR分解方法与该联合优化框架相结合以获得低秩近似矩阵，缩小了权重矩阵和低秩近似矩阵之间的差距以压缩图像分类任务上的DNN模型；最后，通过枚举求解最优秩并迭代地获得具有低秩特征的压缩结果. 模型蒸馏可以将一个复杂的模型（教师模型）转化为一个更小、更简单的模型（学生模型）. Chen等人[22]提出了一个新的框架，即使用知识蒸馏和提示学习来学习紧凑且快速的目标检测网络，并通过例如用于解决类别不平衡的加权交叉熵损失、用于处理回归组件的教师有界损失和用于更好地从中间教师分布中学习的适应层等创新来解决检测的复杂性提出的挑战. Mishra等人[23]提出了一种新方法EarlyLight，旨在通过基于知识蒸馏的训练程序设计和训练轻量级DNN，以满足边缘设备上任务执行所需的存储、处理速度和最大允许处理时间限制，并引入了提前停止技术来加快训练过程，从而解决了在边缘设备上部署DNN时所面临的算力、能源和存储需求过高的问题. 为了解决现有剪枝方法在修改损失函数和复杂训练过程中的问题，施瑞文等人[24]提出了一种基于几何距离度量通道间相关性的特征导向剪枝方法，通过贪婪策略优化滤波器剪枝，实现在剪枝与网络结构、训练过程之间的解耦，从而简化了剪枝任务. Basha等人[25]提出了基于历史训练信息的滤波器剪枝方法. 该方法通过分析滤波器权重的绝对值和在训练过程中的变化模式，识别冗余滤波器并进行剪枝，从而显著减少计算量和存储需求，同时保持较低的错误率.

然而，这些模型压缩技术旨在减少模型参数数量和大小或计算量（MACs），并未关注到能耗、时延等系统性能. 另外，以上所有模型压缩技术都需要经过多次的模型重训练来获得较高的准确度，从而无法本地在线对深度学习任务进行预测. 本文提出的可动态伸缩的移动端深度计算图算优化方法提前将模型及压缩算子变体在云端离线训练，本地部署后对深度学习任务保持在线响应，并考虑利用移动设

备的资源来指导压缩算子变体的选择.

1.2 算子融合技术

算子融合是基于计算图中间结构的移动深度学习框架中的重要编译优化技术，例如 MNN[26]，TVM[27]. 算子融合遍历和搜索整个计算图，按照一定的规则匹配并将计算图中相邻的“生产者”算子和“消费者”算子融合成新的算子以增强中间数据的存储访问局部性. 算子融合技术能够有效地减少中间计算图推理过程对内存的访问次数和访存时延，从而降低模型推理的时延与能耗，克服内存瓶颈问题. 现有研究[28]和深度学习框架(例如 Pytorch[29])已经很好地探索了各种基于固定规则的算子融合技术. 然而这些固定的融合过于受限，无法覆盖不断演进的新深度模型中的不同算子和层连接，例如，BERT 模型中的 Conv-Transpose + ReLU + Concat、GPT-2 模型中的 MatMul + Reshape + Transpose + Add，以及 TinyBERT 模型中的 Sub + Pow + ReduceMean + Add + Sqrt 等规则无法在 TVM 中被识别和融合，从而导致次优的加速性能. 由于严格基于规则的算子融合不能保证性能(例如时延)始终是最佳的，为了解决这个问题，MetaFlow[30]提出基于一组宽松图替换规则的回溯搜索算法来优化算子融合的搜索成本，并使用基于流的图分割算法将完整计算图空间递归地分割成更小的子图搜索空间，以实现局部高效搜索与融合优化. 随后，Zhou 等人[31]提出了一种基于深度强化学习的可迁移优化器来优化搜索策略以进一步提高优化效率. 与以前的方法(例如分层设备布局(HDP)[32]、Spotlight[33]、Placeto[34])不同，它基于整个计算图而不是单独的图节点做出决策，大大加快了搜索速度. 为了解决已有方法中融合规则与搜索策略粗糙低效、代码生成能力有限等问题，Zheng 等人[35]提出了深度学习编译器 Fusion-Stitching，它能够将具有不同数据依赖性和非同构并行性的内存密集型运算符融合到大型 GPU 内核中，以自动减少全局内存访问和上下文切换开销. 随后，Hu 等人[36]提出了编译式深度学习框架 Jittor，它提供了一些类似 numpy 的元算子并集成了增强的算子融合规则. 为了优化 Transformer 模型在运行时由于数据读取和写入而带来的传输时延，Ivanov 等人[37]分析了 Transformer 中常见的 3 类算子(张量收缩算子、统计归一化算子、逐元素类算子)对数据传输的特定影响，并以减少数据冗余传输为目标，识别每种算子减少数据移动的机会并针对 Transformer 设计和应用了新的融合规则. 为解决现有方法在算子融合规则和搜索策略上的局限性，DNNFusion[38]提出了一种基于算子视图的融合框架，通过分类和组合分析扩展融合机会. 该框架引入数学性质的图重写和优化的代码生成，提供了一种更灵活和高效的算子融合方法，克服了传统方法在深度模型中应用的局限性. 为了解决内存密集型计算导致现有方法无法在复杂 2 级依赖关系下结合即时需求进行高效算子融合的问题，Zheng 等人[39]系统地抽象了 4 种算子融合方案并考虑了多维优化目标，提出 AStitch. AStitch 通过新的分层数据重用处理复杂的计算图依赖并通过自适应线程映射高效地处理各种矩阵形状，为内存密集型计算探索了新的算子融合优化空间. 谢坤鹏等人[40]提出了 SAF-CNN 框架，其中包括计算图重构和硬件算子融合优化方法，解决了传统卷积神经网络加速器在 FPGA 上部署时算子适配和调度问题. 该方法通过对计算图进行重构，并融合硬件算子，提高了目标检测任务的模型推理速度. 为了提升数据流架构在 YOLO 神经网络中的执行效率，穆宇栋等人[41]提出了数据流图层次上的算子融合调度机制. 该机制充分利用了 YOLO 神经网络中算子间的结构耦合特性，通过融合调度减少数据存取次数，从而有效提高了神经网络的整体运行效率.

现有的算子融合方法主要在编译阶段对计算图进行独立优化. 然而上层模型在运行时由于情境变化而表现出模型结构的动态性，这些方法需要重复搜索新编译的计算图以实现在线算子融合的优化，这导致了计算图的冗余搜索和次优融合加速效果. 本研究提出的方法能够适应多种算子融合规则，它在设备运行时考虑到融合规则带来的加速效益和搜索开销，以此指导上层模型结构的优化. 这使模型压缩时生成更紧凑的、与融合规则高度匹配的算子结构，实现上层模型压缩和底层编译方法的跨层协同和运行时模型轻量化.

1.3 内存优化技术

内存优化技术通过有效管理和减少内存占用、提高内存利用率来提升系统性能，特别是在资源受限的移动终端环境中. 该技术通常分为基于底层数据存储的直接操作和基于上层计算图结构的修改 2 种类型. 前者包括内存交换技术和内存分配技术. 内存交换技术指的是在前向传播过程中，将暂时不需要的中间激活数据从 GPU 等有限内存转移到更大的内存，比如 CPU. 当反向传播中梯度更新过程需要这些中间激活数据时，它们可以再次被交换回 GPU 内存进行使用. Rhu 等人[42]提出了一个运行时内存管理器，通过在训练过程中实现 GPU 和 CPU 内存的并发

使用来虚拟化模型的内存利用率，这个方法能够将多个主流型号的 GPU 平均内存使用率降低 90%. Ren 等人[43]提出了 Sentinel，这是一个针对异构内存的运行时系统，能够自动优化张量管理. 其核心思想是通过将生命周期和内存访问频率相似的张量分配到同一内存页面，实现最佳的内存共同分配，避免不必要的数据移动. Jung 等人[44]提出了 DeepUM，旨在通过利用 CUDA 统一内存和 CPU 内存作为 GPU 的后备存储. 他们在 UM 块级别使用了一种新的相关预取机制来隐藏错误处理开销，这大大减少了内存交换时间，并增加了 GPU 的虚拟内存容量. 为了解决传统页交换机制在低延迟 SSD 环境下的瓶颈，王紫芮等人[45]提出了一种优化的超低延迟 SSD 页交换机制（Ultraswap）. 该机制通过在 Linux I/O 栈中增加对轮询请求的处理，并降低 I/O 合并与调度开销，优化了页交换的 I/O 栈，减少了内核在换入和换出路径上的时间开销. 内存分配技术是指如何有效地分配系统内存以支持 DNN 的训练和推理过程. Moon 等人[46]分析了训练过程中出现的所有张量，其中包括输入数据、权重和中间激活，根据这些张量的生命周期实现了不相交张量的内存复用，大大减少了内存的使用. Wang 等人[47]提出的 Melon 通过感知张量生命周期的算法，优化了内存布局，基于贪婪方法将长周期张量有限分配在低地址空间，以减少内存碎片并提升内存利用率. 张传奇等人[48]提出了 GroupUCP，一种按需调节的细粒度缓存资源管理方法. 通过动态分组和实时评估，GroupUCP 根据每个应用对缓存组的不同需求，允许各缓存组独立进行路划分分配，从而提高了缓存利用率并优化了系统性能，特别是在缓存资源需求不均衡的应用场景下，能够有效减少资源浪费并提升整体运行效率.

基于上层计算图结构的修改包括了重计算和算子重排 2 种技术. 重计算技术在前向传播计算结束后立即丢弃部分中间激活来节省内存使用，并在反向传播时重新计算丢弃的激活以进行梯度计算. Chen 等人[49]提出了重计算技术，该技术将模型分解为几个部分，在正向计算后仅保留每个部分的首次激活. 在反向传播过程中，从保留的首次激活开始计算每个部分中的激活. 这项工作实现了显著的内存节省，并且对于计算时延和内存空间之间的权衡是有益的，特别是在有足够计算资源的情况下. Zong 等人[50]提出了一种名为 STR 的新型混合张量重新生成策略，结合了内存交换和重计算技术，旨在在内存有限的情况下找到 DNN 训练的最佳执行计划. 算子重排技术是指将模型中算子的拓扑顺序按照某些规则进行重新排布，在不降低模型的推理精度的前提下，降低模型的内存占用[51]. Lin 等人[52]提出了微型训练引擎（TTE）来重新排序深度学习训练中的算子. TTE 通过调整反向传播的算子执行顺序，在每次更新第 i 层后立即丢弃第 i 层的梯度，而不是保留整个训练迭代期间的梯度. 这样做可以有效节省内存，因为立即释放无用张量所占用的空间有利于内存的节省. Steiner 等人[53]用神经网络逐层计算的可预测性，基于整数线性规划对算子重新排序，优化了计算图中数据张量的生命周期，从而实现了更少的内存碎片和更低的训练内存占用.

上述技术中大部分关注的是训练过程中反向传播的设计，难以推及到推理过程中运用. 本文提出的方法通过引入计算图分支的拓扑重排和逐元素类算子的重排，探索内存占用最少的算子执行拓扑排序，从而协同实现了模型运行时推理过程中的内存优化.

1.4 自适应深度模型动态计算

自适应深度模型动态计算在设备运行时，通过动态调整模型结构或推理路径实现对原始部署模型的选择性执行，从而优化深度模型运行时的计算开销. 1）动态模型结构. Han 等人[54]自适应地选择模型的不同变体以优化模型准确性并满足移动设备资源限制（例如内存和能量）. Yang 等人[55]使用不同的输入分辨率训练一组具有不同宽度的子网，以在输入分辨率发生变化时自适应选择匹配宽度的网络，减少低分辨率输入的计算冗余. Marco 等人[56]基于推理任务的精度需求和时延约束，动态确定和匹配当前输入下最优的网络变体结构并执行推理. 2）动态推理路径. Wu 等人[57]在推理过程中，基于强化学习训练的策略网络自适应选择参与计算的网络残差块，以在不降低模型精度的情况下优化模型推理计算量. Teerapittayanon 等人[58]发现网络的浅层特征足以对目标数据进行分类，因此提出一种允许前向传播计算从模型浅层分支早退出的多分支网络，以降低前向传播的计算成本. Gao 等人[59]提出特征增强和抑制方法，利用卷积层计算的特征重要性来预测和增强重要的卷积通道并跳过不重要通道的计算. Yang 等人[60]提出了基于强化学习训练的一次跳过模块，以有效地控制 DNN 模型计算中的块跳过. Sun 等人[61]提出了一种基于动态推理的方法. 该方法通过分析中间层的推理能力，发现仅使用第 1 层就能准确生成 54.4% 的标记，从而显著降低了计算量. 基于此，研究者进一步提出了一种新的动态推理机制，旨在减

少不必要计算的同时提高准确度.

然而,这些方法高度依赖推理路径和网络变体结构的预定义设计空间.在将模型实际部署到未知的动态移动环境之前,这些方法难以预先设计匹配的选择空间.本文通过动态选择并组合适当的压缩算子变体,在运行时从多个缩放维度灵活压缩模型,实现对未知动态移动环境的实时弹性伸缩计算.

2 研究挑战与动机

运行时自适应的DNN跨层优化框架是实现移动端高效推理的理想方法,但在资源有限的异构移动设备和复杂多变的运行情境中,算法和系统层面临挑战.具体表现为设备资源和系统的异构性以及运行时情境的动态性,使得面向固定资源和性能需求的部署方案在不同设备和运行时情境下难以实现最优效率.

2.1 移动终端资源与系统异构性导致预定义模型部署效率难以达到最优

移动终端设备在深度学习领域中扮演着越来越重要的角色,但由于其硬件资源和系统的异构性,固定的部署方案在不同设备上难以实现最优效率.这种异构性主要体现在计算能力、存储资源和能源效率等方面,如表1所示,以Jetson Xavier NX和Raspberry Pi 4B为例.

Table 1 Hardware Resources and System Environments of Different Devices are Heterogeneous

表1 不同设备的硬件资源和系统环境异构

性能指标	Jetson Xavier NX	Raspberry Pi 4B
单精度浮点算力/GFLOPS	1 300	5
功耗/W	10	5
内存	8 GB LPDDR4x	2 GB/4 GB LPDDR4
单次浮点计算能耗/mJ	0.48	3.1
单次访存能耗/mJ	1.5	2.8

2.1.1 移动终端设备的计算能力不同

不同移动终端设备的处理器性能差异显著,导致固定的部署方案难以在所有设备上优化运行:1)处理器性能差异.不同设备的处理器时钟频率和核心数量各异.表1显示,Jetson Xavier NX的计算能力比Raspberry Pi 4B高出10^4个数量级.固定的部署方案若针对高性能处理器设计,在低性能处理器上运行会导致计算速度显著降低;反之,高性能处理器上资源将无法被充分利用.2)架构与指令集差异.不同处理器支持的指令集和硬件加速功能不同.固定的部署方案若未针对特定指令集优化,在支持这些功能的处理器上运行时无法发挥最佳性能.3)多核利用差异.移动终端设备的多核处理能力各异.未能充分利用多核进行并行计算的固定部署方案,在多核设备上无法实现性能最大化,而在单核设备上则可能导致资源竞争以及性能下降.

2.1.2 移动终端设备的存储资源不同

存储资源的异构性直接影响模型的部署效率:1)不同设备内存容量差异巨大.如表1所示,Jetson Xavier NX的内存容量比Raspberry Pi 4B大几倍.固定的部署方案所预估的DNN内存占用若小于设备内存,会导致内存未充分利用,计算效率低;若超出设备内存,则会导致内存溢出和频繁的内存交换,从而降低性能.2)设备之间的闪存容量不同.大模型在小容量设备上可能无法存储,从而增加了数据传输的时延;而在大容量设备上,未充分利用存储空间而导致资源浪费.3)不同设备的存储速度不同.固定的部署方案在低速存储介质上运行时,数据读取和写入速度慢,影响模型的加载和推理效率;然而在高速存储介质上若未能充分利用其优势,数据处理效率同样会受到限制.

2.1.3 移动终端设备的能源效率不同

能源效率差异显著影响模型部署的实际效果:1)电池容量差异.设备间的电池容量不同.高负载计算任务部署在小容量电池设备上将会快速耗尽电量,影响其运行的连续性;而在能源充足的设备上过度优化能耗使得资源浪费,从而导致性能下降.2)计算与访存能耗差异.由于不同设备的处理器架构、芯片制程、内存总线配置、缓存层级、带宽等计算和内存结构的差异,这些设备的单次浮点计算和单次访存能耗不同,如表1所示.因此,对于单位计算或访存能耗高的设备,需要对模型中计算密集或访存密集的算子进行针对性优化,以最大程度降低模型推理过程中计算或访存能耗.

移动终端设备的计算能力、存储资源和能源效率的异构性,使得固定的部署方案难以在所有设备上达到最优效率.需设计灵活高效的部署方案,利用多变体模型伸缩、自适应算子融合与重排等技术,充分考虑设备资源和系统的差异,提升移动计算的整体性能.

2.2 运行时情境动态变化导致模型计算效率低

移动终端设备由于其接近用户和实际应用场景的特性,面临数据、环境和资源实时动态变化的挑

战. 固定的部署方案在这种环境下难以达到最优效率, 主要原因在于它无法灵活适应输入数据、可用资源和性能需求的变化.

2.2.1 移动计算数据分布差异

数据分布的动态变化对固定部署方案提出了巨大挑战: 1)输入数据的规模和复杂度可能会随时间和应用场景的变化而波动. 例如, 在自动驾驶中, 高清摄像头每秒捕获 30 帧, 每帧 12 MB, 每分钟数据量约为 21.6 GB; 而普通监控摄像头每秒捕获 15 帧, 每帧 1 MB, 每分钟数据量约为 0.9 GB. 固定的部署和压缩方案难以实时适应数据复杂度的变化并高效处理复杂数据. 2)输入数据的到达频率随着传感器的采集频率动态变化. 例如, 自动驾驶车辆中的激光雷达数据采集频率在城市道路和高速公路上存在差异. 城市道路上每秒采样 20 次, 高速公路上每秒采样 10 次. 数据到达频率高时, 固定的部署方案无法及时提高模型推理频率, 导致时延增加. 特别是在高频率数据输入场景中, 若模型无法迅速响应和处理, 将可能造成数据积压和处理时延, 进而影响系统的实时性和用户体验. 相反, 在低频率数据输入时, 固定的部署方案可能无法充分利用可用资源, 导致计算资源闲置和效率低下.

2.2.2 移动计算资源实时变化

资源需求的动态变化进一步增加了固定部署方案的挑战: 1)不同时间段运行的应用程序和任务对内存资源需求波动较大. 例如, 在智能家居系统中, 白天家电设备使用频繁, 多个任务同时执行抢占内存资源, 对内存资源需求较大; 而夜间设备使用减少, 从而不需要太多内存资源. 固定的部署方案若未能动态调整内存使用策略, 可能在内存资源紧张时导致内存溢出和性能下降, 在内存资源充足时未能充分利用内存, 造成资源浪费. 2)移动终端设备需要同时处理多个任务, 计算资源需求实时变化. 例如, 自动驾驶在复杂的城市道路环境计算负载较高; 而在空旷的高速公路上仅需要低计算负载. 固定的部署方案如果未能动态分配计算资源, 在高负载时会导致 DNN 计算效率低下, 任务响应时间增加; 在低负载时, 计算资源闲置, 未能高效利用. 3)移动终端设备依赖电池供电, 电量随设备使用情况实时变化. 例如, 无人机在长时间飞行任务中, 电量从 100% 降至 20%, 需要调整策略以降低能耗. 固定的部署方案若未能动态调整能耗策略, 在电量低时难以维持高负载任务, 导致设备续航能力不足; 在电量充足时优化能耗, 浪费电池资源. 4)设备运行环境的温度变化会影响硬件性能和稳定性. 移动终端设备在夏季室外环境温度高达 40 ℃ 时, 处理器温度升高, 可能导致降频甚至关机; 在冬季低温环境下, 处理器温度低于 0 ℃, 电池性能下降 30%. 固定的部署方案未能考虑温度变化对设备性能的影响, 导致在极端温度下表现不佳.

2.2.3 DNN 计算的动态性能需求

上层任务对 DNN 计算性能的需求动态变化, 固定的部署方案难以适应, 包括以下 3 方面因素: 1)不同应用场景对 DNN 计算精度要求不同. 例如, 在自动驾驶应用中, 夜间或恶劣天气下的行车识别任务需要更高的精度以确保安全, 这时对模型精度的要求可能比白天晴朗天气高出 20%. 固定的部署方案可能导致在高精度场景下计算能力不足, 低精度场景下资源浪费. 2)实时应用对计算时延的要求随移动场景不同而变化. 例如, 在实时视频处理应用中, 为了保证流畅的用户体验, 视频流处理的时延通常要求低于 100 ms, 而在非实时的视频分析中, 时延可放宽至 500 ms 以上. 固定的部署方案如果未能实时优化模型结构、计算图拓扑和资源分配, 可能导致高时延影响应用效果和用户体验. 3)移动终端设备中的内存预算随着系统中其他竞争进程的变化而波动. 例如, 在智能手机中, 当用户同时运行多个应用时, 系统的可用内存可能从 4 GB 降低到 1 GB, 这会导致某些内存密集型模型的性能显著下降. 固定的部署方案若未能动态分配内存资源, 可能在内存预算低的情况下导致性能下降, 而在内存预算高的情况下未能充分利用内存资源.

移动终端设备在移动计算中面临运行时数据差异、资源变化和性能需求等动态影响因素, 使得固定的部署方案难以在所有情况下达到最优效率. 基于当前运行时条件, 精准构建跨层方案空间, 快速搜索跨层部署方案, 通过实时优化和自适应调整技术应对边缘环境多变性来确保模型计算的资源高效性.

3 基于可伸缩图算结构的跨层优化

受设备运行应用程序、环境变化以及任务需求波动的影响, 移动设备的可用资源(如存储、计算能力和电池)具有动态性. 因此使设备端模型具备可伸缩性以应对资源波动, 从而确保在动态环境下维持计算效率和实时性变得尤为重要. 现有的深度学习模型压缩方法通常生成静态的压缩模型, 无法在运行时根据设备资源的波动进行动态调整, 这限制了

它们在高度受限的移动终端设备上的部署性能. 此外, 一些传统的静态优化方法往往只能在模型结构层面进行压缩, 而无法同时实施计算图层和内存层的协同优化, 导致在实际应用中难以应对设备资源的动态波动, 尤其是在高计算需求和严格的资源限制下.

为了解决这一问题, 本节提出了一种基于可伸缩图算结构的解决方案. 与传统单层压缩方法不同, 该图算结构通过引入可替换压缩模块、可融合计算图和可重排算子序列, 与预训练的骨干网络结合, 构建了一个可伸缩的跨层结构. 该结构的可伸缩性涵盖模型层-多变体可伸缩模型结构(3.1 节)、编译层-模型变体适配的计算图(3.2 节)、内存层-内存感知的计算序列调度(3.3 节), 如图 3 所示. 该结构支持在运行时根据硬件资源的变化自动调整模型层压缩、计算图层融合和内存层调度的策略, 快速实现伸缩并在动态环境下维持模型性能. 具体来说, 模型层压缩变体与计算图层拓扑结构的设计紧密耦合, 优化了算子的融合模式与执行顺序, 利用了基于硬件特性的动态算子重排和多级内存调度策略, 从而实现了高效的资源调度和计算优化. 通过跨层协同设计, 模型能够根据设备的资源状态自适应调整, 从而在动态环境中保持最佳性能.

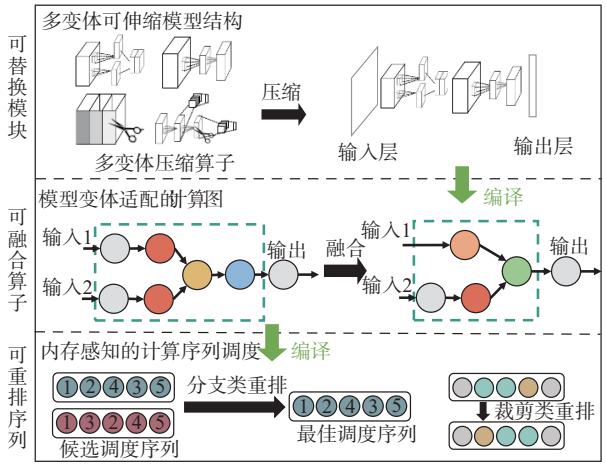


Fig. 3 Scalable graph computing architecture

图 3 可伸缩图算结构

3.1 多变体可伸缩模型结构设计

尽管深度模型的压缩算法取得了显著进步[62-63], 但目前的方法往往只能一次性生成固定的压缩后模型. 这些方法缺乏在持续运行的移动应用程序环境中进行动态上下文适应的自动化集成能力. 一个关键挑战在于如何在运行时动态地调整深度模型的压缩水平, 以适应不断变化的环境条件. 传统的压缩技术通常不支持在运行时轻易地重新扩展已压缩的模型, 从而限制了它们在实际应用中的灵活性. 为了解决这一问题, 我们引入了一种预训练的多变体算子可伸缩模型. 该模型为骨干网络配备了多个压缩算子变体, 使其能够根据运行时条件的变化动态调整压缩水平. 这种方法不仅提高了模型在资源受限环境中的适应能力, 还保留了良好的性能和准确性. 通过集成多个压缩算子变体, 模型可以灵活地在不同的压缩状态之间切换, 从而在保证计算效率的同时, 最大限度地满足应用程序的需求.

我们选择了近年来在轻量化网络结构设计中被广泛使用和验证的高效运算模块作为我们的压缩算子变体. 这些模块用于替换骨干网络中的传统卷积算子, 通过调整模型的宽度、深度和连接性等结构特性, 优化其运算模式, 从而影响模型在计算平台上的实际性能和资源消耗. 接下来, 本节将介绍这些压缩算子变体及其原理.

3.1.1 Bottleneck 结构块

Bottleneck 结构块作为早期轻量化网络模型中最为著名的构建块思想, 已经在 Inception[64] 和 ResNet[65] 等经典网络中得到了广泛应用. 其核心思想是将传统的卷积操作分解成多个更小的卷积操作, 从而实现模块的轻量化. 例如, 传统的 3×3×256 卷积操作在 Bottleneck 结构中被替换为一个 1×1×*num*、一个 3×3×*num* 和另一个 1×1×256 的卷积运算, 如图 4(a)所示, 其中压缩率 *num* 可以设置为任意小于 256 即输出维度的正整数. 这 2 个 1×1 卷积运算分别用于降低和升高特征维度, 而中间的 3×3 卷积则用于特征提取.

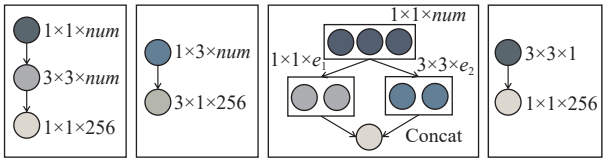


Fig. 4 Four types of compression operators

图 4 4 类压缩算子

通过这种分解结构, 模型的深度得以加大, 但参数量和计算量却同时显著减少. 具体来说, 1×1 卷积操作大幅减少了输入特征的维度, 从而降低了随后的 3×3 卷积的计算复杂度; 在输出端, 另一个 1×1 卷积则将特征维度恢复到原始水平. 因此, Bottleneck 结构在保留模型性能的同时, 实现了高效的参数和计算量压缩.

为了适应不同的压缩需求, 我们将 *num* 的取值分别设置为输出维度的 25%, 50%, 75%, 代表不同的

压缩率. 通过这种方式, 可以灵活调整模型的压缩水平, 以满足不同计算平台和应用场景的性能要求和资源限制.

3.1.2 低秩分解结构块

低秩分解结构块采用了一种低秩近似技术, 通过将传统卷积操作分解成多个卷积操作来实现参数和计算量的降低. 这种方法包括二分量分解、三分量分解或四分量分解等形式.

我们使用一种二分量分解方法, 即奇异值分解(SVD). 具体来说, 传统的 3×3×256 卷积运算在低秩分解结构中被分解为 1×3×*num* 和 3×1×256 的 2 层卷积运算, 如图 4(b)所示. 通过这种方式, 模型的深度得以加大, 但参数量和计算量却显著减少. 1×3 卷积操作先将输入特征在水平方向上压缩, 而 3×1 卷积则在垂直方向上进行特征提取. 因此, 计算复杂度得以降低, 同时保留了卷积操作的特征提取能力.

我们同样将 *num* 设置为输出维度的 25%, 50%, 75%, 分别代表不同的压缩程度, 从而根据不同情境选择合适的压缩比例.

3.1.3 Fire 结构块

Fire 结构块源自 SqueezeNet[66], 这是最早期的轻量化网络模型之一. 作为该模型的核心创新, Fire 模块结合了 Bottleneck 结构块和多分支并行结构块的特点. Fire 模块由 Squeeze 部分和 Expand 部分组成, 其中 Squeeze 部分是一组连续的 1×1 卷积, Expand 部分则包含一组连续的 1×1 卷积和一组连续的 3×3 卷积, 两者结合后进行串联, 如图 4(c)所示.

以传统的 3×3×256 卷积运算为例, Fire 结构块首先通过 Squeeze 操作, 即使用 1×1×*num* 的卷积对输入特征进行降维. 然后, 通过 Expand 操作, 对降维后的特征分别进行 1×1×e_1 和 3×3×e_2 的卷积运算, 最终将 2 种卷积的输出合并, 使得 Fire 块的输出维度等于 e_1+e_2.

我们采用了 2 种 Fire 结构块的配置方式. 首先, 将 *num* 设置为整个结构块预定输出维度的 1/8(0.125)和 1/4(0.25); 其次, 将 e_1 和 e_2 设置为相同的大小, 均为最终输出维度的一半. 通过这种配置, Fire 模块能够在保持高效压缩的同时, 灵活调整模型的结构.

3.1.4 深度可分离卷积块

深度可分离卷积块(depthwise separable convolution, DSC)由 Google 公司的 MobileNet[67] 首次提出. 这种结构与 Fire 模块类似, 采用了 2 阶段的卷积运算模式, 即深度卷积和逐点卷积. 然而, 与前述方法不同, 深度卷积只对单个通道(即输入特征的 1 个维度)进行卷积, 而不是像标准卷积那样对所有输入通道进行操作. 如图 4(d)所示, 可以理解为深度卷积的卷积核维度为 1, 且卷积核数量等于输入特征的通道数. 逐点卷积则是 1×1 卷积, 用于调整深度卷积后的特征维度, 类似于 Bottleneck 结构中的升降维操作. 深度可分离卷积的优势在于, 它将卷积操作的特征深度和宽度分离开来, 大大降低了参数量和计算量. 通过这种设计, 深度可分离卷积块在保持模型性能的同时, 显著减少了计算资源的消耗, 使其适用于资源受限的计算平台和移动设备. 基于上述压缩技术, 我们构建了不同配置的压缩算子变体, 作为结构块组成具有不同深度、宽度、连接性和运算模式的深度学习模型来适应异构设备和动态性能需求. 我们将这些多样化的算子结构块集成到骨干模型中, 作为实现模型自适应机制的基础和跨层策略空间的模型结构子空间.

3.2 模型变体适配的计算图层策略空间

3.1 节介绍了多变体可伸缩模型结构, 其在模型结构上进行了轻量化设计, 使得模型可以更好地部署在资源受限的设备上. 本节基于上层模型和底层指令间的中间结构(计算图)进行优化, 根据多变体可伸缩模型结构进行算子融合机制设计, 进一步优化模型的性能, 并且此节提出的算子融合机制也为后续自适应策略搜索空间提供了基础.

计算图中相邻算子节点间数据流的输入输出会显著增加内存使用和数据访问时延. 为了解决这个问题, 将多个相邻算子合并为 1 个算子是一种有效的方法. 尽管 Mace[68] 等深度学习框架提供了相关功能, 但在多变体压缩模型上的优化效果有限. 我们基于不同多变体模型中的常见算子类型, 提出了 One-to-One, Many-to-Many, Reorganize 的算子融合策略, 并动态调整融合模式来提高效率. 这能够减少中间特征的访存开销和需要管理执行的独立算子数量来降低时延.

本文基于卷积层和批归一化层融合原理, 在 3.1 节设计的多变体可伸缩模型结构上将计算图中识别的特定结构进行融合. 这种方法不仅提升了模型性能, 还减少了部署和运行的复杂性, 确保模型能够在资源受限的环境中高效运行.

本文涉及到的算子操作如表 2 所示, 可以看出, 所涉及的算子操作有非常多的融合机会. 我们将涉及的算子操作视为一个全集 U={BatchNormalization, Concat, ReLU, Conv, MaxPool, Softmax, Flatten}, 基于全集 U 进行算子融合策略的设计.

Table 2 Typical Operator Operations

表 2 典型算子操作

骨干模型	压缩算子变体	算子操作
AlexNet VGG16 ResNet18 ⋮	Bottleneck 结构块 SVD 结构块 Fire 结构块 DSC 结构块 ⋮	BatchNormalization, Concat, ReLU, Conv, MaxPool, Softmax, Flatten, …

我们根据输入和输出之间的映射关系将全集 U 中的算子操作进行分类, 分类结果如表 3 所示. One-to-One 表示输入张量上的第 i 位与输出张量上的第 i 位存在一对一的映射关系; Many-to-Many 表示输入张量上的多位与输出张量上的多位存在映射关系; Reorganize 也是一对一的映射, 但与 One-to-One 不同, 输入张量的第 i 位可能映射到输出张量的第 j 位.

Table 3 Operator Operation Classification

表 3 算子操作分类

映射类型	算子操作
One-to-One	BatchNormalization, Concat, ReLU
Many-to-Many	Conv, MaxPool, Softmax
Reorganize	Flatten

完成基础算子分类后, 需要确定融合策略.指定融合策略时, 只需考虑相邻的 2 个算子是否能融合. 一旦 2 个相邻算子形成一个融合算子, 将其视为一个新的独立算子, 并递归评估该融合算子与前后相邻算子是否可以进一步融合. 定义融合后形成的新算子类型: 如果算子 A 和 B 同属于类型 x, 则融合后的 AB 仍为 x; 如果 A 属于 x, B 属于 y, 则选择 x 和 y 中较复杂的类型作为 AB 的类型.具体来说, One-to-One 的复杂度最低, Reorganize 居中, Many-to-Many 最高.

具体转换关系见表 4, 白色区域表示一定可以融合, 深色区域表示 Many-to-Many 与 Many-to-Many 一定不能融合, 斜条纹区域表示融合后收益不确定, 原因如下:

Table 4 Analysis of the Possibility of Operator Fusion for Different Types of Mapping

表 4 不同映射类型的算子融合可能性分析

算子类型 1	算子类型 2		
	One-to-One	Many-to-Many	Reorganize
One-to-One	One-to-One	Many-to-Many	Reorganize
Many-to-Many	Many-to-Many	×	Many-to-Many
Reorganize	Reorganize	Many-to-Many	Reorganize

1)One-to-One 类型与其他类型融合可以减少多余的数据拷贝, 占用较少的寄存器, 并带来收益.

2)Reorganize 类型仅在 One-to-One 的基础上增加了点对点映射函数, 本质未变, 理由同上. 然而, 与 Many-to-Many 类型融合时, 可能引入数据复制、访问顺序更改或冗余计算, 因此需要进一步分析以验证是否有收益.

3)Many-to-Many 类型与 Many-to-Many 类型融合(如 Conv+Conv)过于复杂, 会影响缓存和寄存器的合理使用, 导致性能下降.

综上所述, 本文提出的多变体可伸缩算子融合机制是基于上述映射机制得到的. 在计算图层面, 通过匹配映射类型, 进行数学属性的图重写, 并计算优化后模型的多个指标, 构建出模型变体适配的计算图层策略空间, 构成跨层自适应搜索空间的基础.

3.3 算子自适应的内存调度层策略空间

本节提出了一种算子重排优化方法, 以在不影响模型精度和计算图结构的情况下优化内存使用. 该方法通过调度算子位置优化内存使用, 包括 2 个模块: 计算图分支的拓扑重排和逐元素类算子的重排. 这 2 个模块可以独立或联合优化模型, 实现更大的性能提升. 前者通过调整分支执行顺序优化计算图结构, 后者则通过前移该类算子位置减少冗余数据传递和存储.

3.3.1 计算图分支的拓扑重排优化

神经网络的计算图中包含分支, 因此在算子的求值顺序上具有一定的灵活性. 当计算到达分支点时, 系统需要决定下一步计算哪个分支, 这一决定会影响内存中的张量存储, 从而影响整体内存占用和计算效率. 通过构建一个执行计划来最小化内存占用, 可以减少内存占用瓶颈并提高计算效率. 例如, 如图 5 所示, 计算图的默认执行顺序是从 op1 到 op7, 但这并不是最佳的执行顺序. 假设 op1 产生的张量是 1 号张量, 默认顺序下, 执行 op3 时需要同时存储 1 号张量、2 号张量和 3 号张量. 通过改变执行顺序为 op1→op5→op6→op2→op3→op4→op7, 可以减少内存占用. 在表 5 和表 6 中, 对比了 2 种执行顺序的内存占用, 结果显示新的顺序在分支处的内存占用减少了约 16%.

这证明了在分支处调整执行顺序可以有效降低内存占用, 同时表明分支选择具有优化空间. 在实际应用中, 分支点可能有多条路径可供选择. 为了找到内存占用最少的执行顺序, 我们采用动态规划算法枚举所有执行计划, 如算法 1 所示, 并计算其内存占用,

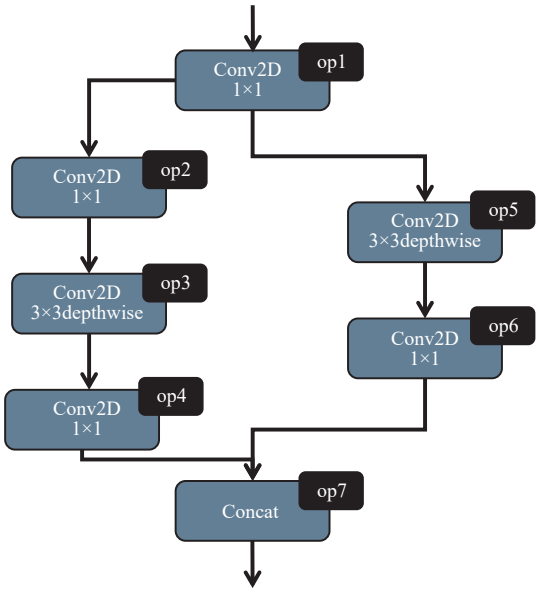


Fig. 5 Illustration of operator reordering in computation graph branches

图 5 计算图分支的算子重排示意图

Table 5 Memory Analysis for Default Execution Order

表 5 默认执行顺序的内存分析

算子	内存中保留的张量	内存占用/B
op1	[0, 1]	6 021
op2	[1, 2]	6 021
op3	[2, 3, 1]	6 676
op4	[3, 4, 1]	5 324
op5	[1, 5, 4]	5 222
op6	[5, 6, 4]	1 434
op7	[4, 6, 7]	1 311
分支平均内存占用/B		4 573

Table 6 Memory Analysis for Execution Order after Topological Reordering Optimization

表 6 拓扑重排优化后的执行顺序内存分析

算子	内存中保留的张量	内存占用/B
op1	[0, 1]	6 021
op5	[1, 5]	4 669
op6	[5, 6, 1]	4 997
op2	[1, 2, 6]	6 349
op3	[2, 3, 6]	2 078
op4	[3, 4, 6]	1 536
op7	[4, 6, 7]	1 311
分支平均内存占用/B		3 852

从而为任意计算图找到内存占用最少的执行计划.

在复杂计算图中，尽管动态规划能够找到最优解，但其计算量通常较大，特别是在多路径选择和算子调度时，搜索过程可能消耗大量的时间和计算资源. 为提高动态规划算法的效率，本研究引入了预剪枝和动态剪枝策略，以减少冗余计算并加速搜索过程. 1）基于内存阈值的预剪枝方法通过静态分析提前剔除不符合内存优化条件的路径. 具体包括：一是内存阈值剪枝，当路径的内存占用超过设备内存限制时，直接排除该路径；二是冗余路径剪枝，通过分析路径中算子的内存依赖关系，剔除那些内存占用较大且无法通过顺序重排优化的路径，避免无效计算. 2）基于路径预测的动态剪枝方法在搜索过程中实时监控路径内存占用，剔除非优路径. 主要策略是内存超限剪枝，当某路径的完整推理图预计内存需求超过内存限制时，立即停止该路径的搜索，避免无效计算. 此外，实时剔除内存占用大于当前最优路径的路径，确保计算集中在内存最优路径上.

算法 1. 基于动态规划的最佳执行顺序搜索算法.

输入：一个在分支处的张量集合以及模型的计算图；

输出：最小内存占用以及计算图执行顺序.

① $\{cs\},\{as\}$, $op_order \leftarrow PARTITION(X, x{:}producer(x)$ is None; /*将张量集划分成$\{cs\}$和$\{as\}$，并提取默认执行顺序，其中常量是指不属于计算图中的算子计算得出的张量*/

② if $\{as\}$ is empty

③ 计算出所有常量的内存和；

④ end if

⑤ 将内存占用设为无穷大，初始化执行顺序；

⑥ for x in $\{as\}$

⑦ $\{rs\} \leftarrow \{as\}/x$; /*取消生成变量 x 的算子*/

⑧ $\{is\} \leftarrow producer(x).inputs$; /*生成变量 x 的算子的输入*/

⑨ if *any* (x is a predecessor of r for r in $\{rs\}$) then

⑩ continue; /*每个算子只执行 1 次*/

⑪ end if

⑫ $m' += \max\left(MEM(\{rs\}\cup\{is\}), \sum_{\{rs\}\cup\{is\}\cup\{x\}} |t|\right)$; /*两者之间取较大的一个作为生成该变量 x 的算子的内存占用值*/

⑬ $op_order = op_order + producer(x)$; /*更新顺序*/

⑭ $m \leftarrow \min(m, m')$; /*选择内存占用小的*/

⑮ end for

⑯ return $\sum_{c\in\{cs\}} |c| + m$, op_order. /*返回内存占用和

执行顺序*/

3.3.2 逐元素类算子的重排优化

逐元素类算子(Element-wise)是一类对输入张量的每个元素进行独立操作的算子,输出张量的每个元素值只与对应位置的输入元素值相关.这类算子可以执行各种数学和逻辑运算,如加减乘除、取负、取反、绝对值等.由于每个元素的操作是独立的,逐元素类算子能够高效并行计算,充分利用硬件资源,提高计算速度.这些算子不受输入张量形状的限制,输出张量的形状与输入张量相同,适用于元素级别的操作.逐元素类算子广泛应用于深度学习模型的各层和模块中,如激活函数、归一化操作、损失函数、优化算法和图像处理等.

算法2展示了逐元素类算子重排技术,其核心思想是在模型中为这类算子寻找最佳执行位置,以减少计算量和存储开销,从而优化推理性能.具体流程如下:首先,在模型中定位逐元素类算子,然后沿着该算子向前移动,遇到第一个非逐元素类算子时停下,该位置即为逐元素类算子的最佳执行位置.

算法2. 逐元素类算子的最佳执行位置搜索算法.

输入:模型的计算图;

输出:计算图执行顺序.

① 遍历计算图,初始化默认执行顺序 *op-order*,
并记录逐元素类算子的位置;

② for *x* in *ops*

③ while *x.predecessor* ∈ *Element_wise*

④ *swap* (*x*, *x.predecessor*);

⑤ *update* (*op_order*); /*遍历每个逐元素类算子,判断当前算子 *x* 的前驱算子是否为逐元素类算子.如果是,则进行位置交换,直到 *x* 前面没有再可以交换的对象,每交换1次都需要对执行顺序进行更新*/

⑥ end while

⑦ end for

⑧ return *op_order*. /*返回最佳执行顺序*/

本节提出的算子重排技术包括计算图分支的拓扑重排和逐元素类算子的重排,可以独立或联合使用.通过灵活调整算子的求值顺序,构建最小化内存占用的执行计划,减少内存瓶颈,提高计算效率.逐元素类算子重排技术为其找到最佳执行位置,减少计算量和存储开销,优化推理性能,构成运行时跨层策略空间中的内存调度子空间.

4 运行时情境自适应跨层策略搜索

在移动端深度学习模型的在线部署中,快速搜索算法-系统跨层优化策略是提升推理效率的关键.由于算法层模型结构、编译层计算图和内存层的调度序列具有显著的差异,如何在算法-系统联合搜索空间中高效选择最优的跨层策略直接影响着部署效率.传统的优化方法通常针对单层空间独立优化,联合这些单层方法进行跨层优化时会面临两大挑战:首先,模型结构、计算图和调度序列在组成单元和结构拓扑上存在显著差异,导致现有方法难以在联合空间中搜索跨层策略;其次,跨层策略组合的数量庞大,搜索空间急剧膨胀,这会导致难以接受的计算开销和高延迟,影响推理部署的实时性.

得益于第3节提出的统一图算结构,本文提出了一种运行时跨层优化策略搜索方法.在4.1节中,我们将模型结构、计算图和调度序列的优化策略空间进行融合,构建了一个精简且统一的跨层搜索空间.通过压缩和合并优化空间,显著降低了搜索复杂度,避免了搜索空间的过度膨胀.在4.2节中,我们提出一种基于进化算法的多目标跨层优化策略搜索方法,能够根据实际情境快速探索最优跨层策略组合.在进化算法中,我们基于高斯过程回归方法通过代理模型预测待选方案的性能,显著减少了搜索时方案性能评估带来的开销.通过引入高斯过程回归与代理模型,使搜索过程能够自适应调整搜索方向从而快速找到最优策略.

4.1 跨层精简搜索空间构建

本文中,情境自适应的搜索空间由第4节介绍的不同骨干网络的压缩算子变体、算子融合机制和算子重排技术组成.由于第3节已详细描述这些内容,本节将对跨层自适应策略的搜索空间进行简要总结.

1)不同骨干网络的选择

选择 M={WideResNet28_10, ResNet18, VGG16}作为骨干网络,因为这些网络结构涵盖了通过宽度提升精度(WideResNet28_10)、深度优化性能平衡(ResNet-18)和堆叠层次捕获特征(VGG16)的多样化设计.在本策略中,骨干网络在部署时已确定,并不参与在线搜索.通过这些多样化的骨干网络,我们可以有效平衡模型精度和性能开销,以适应不同应用场景的需求.

2)不同压缩算子的选择

选择 C={SVD_25%, SVD_50%, SVD_75%, Bottlen-

eck_25%, Bottleneck_50%, Bottleneck_75%, Fire_12.5%, Fire_25%, DSC}作为压缩算子变体的集合，因为这些算子显著减少了模型的参数量和计算量，从而提升效率. 具体来说，Bottleneck 和 SVD 结构块通过设置不同的压缩率(25%, 50%, 75%)有效降低了模型复杂度和内存占用；Fire 结构块以 12.5% 和 25% 的压缩率实现了更高效的计算；DSC 通过将标准卷积分解为深度卷积和逐点卷积，显著减少了计算量和参数量. 这些多样化的压缩算子变体提供了灵活的优化选择，能够在不同应用场景中有效平衡模型精度和性能开销.

3)不同算子融合规则的选择

选择 F={Conv+BN, Conv+BN+ReLU, Concat+ReLU}作为算子融合机制的集合，因为这些融合机制能显著减少中间特征图的内存占用和处理时延. 它们提供了灵活且细粒度的优化选择，能够在不同应用场景中有效控制模型的执行效率.

4)不同算子重排技术的选择

选择 R={计算图分支的拓扑重排优化，逐元素类算子重排优化，联合优化} 作为重排策略的集合，因为这些策略能够显著优化计算图的执行效率和资源利用率. 计算图分支的拓扑重排或逐元素类算子的重排可以灵活适应不同的网络结构和应用场景，提供更高效、优化的计算图执行方案.

上述内容构成了本文的自适应策略搜索空间. 模型结构层面的策略空间为 C，包括压缩算子变体，其策略数为 N_C；计算图层面的策略空间为 F，包括算子融合机制，其策略数为 N_F；内存调度层面的策略空间为 R，包括算子重排技术，其策略数为 N_R. 本文自适应策略搜索空间的总搜索维度为 $N_C \times N_F \times N_R$. 因此，相比于单层优化的设计，本文的自适应策略搜索空间具有更高的多样性，能够在更多维度上实现情境的最优适配.

4.2 多目标跨层优化策略搜索

深度模型是否能成功部署于资源受限的移动终端设备上，不能仅关注模型参数量、计算量等与模型结构和运算模式相关的指标，还需综合考虑推理框架、平台底层支持和硬件因素等. 必须使用模型在实际平台上的开销来指导部署过程. 模型情境自适应正是基于这种综合考虑，本文关注的情境指标如下：

1)存储预算与运行时内存占用

我们将用 M 表示当前使用的骨干网络，c 表示模型选取的压缩算子变体，f 表示当前选取的算子融合规则，r 表示当前选取的算子重排技术，S 表示运行时的内存占用，得到如下策略：

$$S(M,c,f,r) \leqslant S_{\text{bgt}}(t), \tag{1}$$

其中 S_{bgt} 表示当前时间 t 下的剩余内存容量，实际情况下将受到其他应用、外设或内核程序占用的影响而不同.

2)运行时延(*Latency*)和能效(energy efficiency, *EE*).

模型的部署和运行必须符合存储预算等平台资源的情境指标约束. 然而，模型的运行时延和应用能效虽然不是硬性部署约束，但也同样重要，因为它们反映了用户或移动应用的自定义需求. 例如，不同应用可能对响应实时性和续航能力有更高要求，而移动终端设备则常常需要低能耗运行模式. 基于本文的搜索空间，策略方案为：

$$Latency(M,c,f,r) < needed_Latency, \tag{2}$$

$$EE(M,c,f,r) < needed_EE. \tag{3}$$

确定了情境指标后，我们进行情境自适应框架的搜索策略设计. 在本文中，这可以表示为带情境约束的多目标优化问题：

$$a^* = \arg\min_{c\in C, f\in F, r\in R} L(M,c,f,r), \tag{4}$$

$$\begin{aligned} L(M,c,f,r) = L\{&1-Acc(M,c,f,r),\\ &S(M,c,f,r), Latency(M,c,f,r)\}, \end{aligned} \tag{5}$$

$$\text{s.t. } S(M,c,f,r) \leqslant S_{\text{bgt}}(t).$$

式(4)中，a^* 表示当前情境下的策略选择，L 表示带约束的多目标优化函数.

接下来，我们根据多目标优化问题进行搜索策略的设计和优化.

1)搜索策略设计. 由于用户情境的软特性，本文将时延和内存 2 项情境指标与模型精度一起作为优化目标. 为优化模型的跨层自适应策略，本文提出基于进化算法(evolutionary algorithm, EA)的多目标优化框架，即算法 3，旨在高维策略空间中搜索最优策略组合，即 $X=(c,f,r)$. 步骤如下：①种群初始化. 初始种群通过随机生成 N 个个体 $x_i=\{(c_i,f_i,r_i)\}_{i=1}^{N}$，每个个体的元素从相应集合中($c_i\in C, f_i\in F, r_i\in R$)抽取. 例如，初始个体可能为(Bottleneck_50%, Conv+BN+ReLU, 计算图分支的拓扑重排优化). 在初始化时，为避免种群陷入局部最优，本文引入多样性约束，确保 N 个个体的 (c,f,r) 组合覆盖策略空间的关键区域. ②适应度评估. 定义适应度函数 $V(X)=\dfrac{1}{L(X)}$，用于量化个体 X 的综合性能 $V(X)$. ③个体选择. 基于适应度 $V(X)$ 的锦标赛选择机制从当前种群 $\{x_i\}_{i=1}^{N}$ 中挑选个体进入下一代. 每轮随机选取子集 $T\subseteq\{x_i\}(|T|\ll N)$，选择适应度

最大的个体$X^* = \arg\max_{X \in T} V(X)$. ④交叉生成. 通过组合父代的策略元素生成后代. 随机选择 2 个父代$x_p = (c_p, f_p, r_p)$和$x_q = (c_q, f_q, r_q)$, 以概率p_c交换其部分元素生成后代$x_o = (c_p, f_q, r_p)$. 交叉扩展了搜索空间覆盖范围, 同时结合不同策略的优点应对复杂场景需求. ⑤替换个体. 生成的新个体与种群中适应度较低的个体替换, 并采用精英保留策略将适应度最高的个体直接保留至下一代, 避免全局最优解丢失.

通过定义压缩算子变体、算子融合规则和算子重排技术的策略空间, 并结合适应度函数评估模型的延迟、内存占用和精度, 我们成功地构建了一个能够动态调整策略的优化流程. 进化算法通过种群初始化、适应度评估、个体选择、交叉生成和替换个体, 有效探索了各个策略组合, 最终实现了在资源受限环境中的高效自适应部署.

2)搜索策略优化. 为了提高进化算法的效率, 本文提出了一种结合高斯过程回归和动态策略调整机制的自适应策略, 以减少计算开销并提高优化效果, 具体步骤包括以下 2 点. ①采用了高斯过程回归(Gaussian process regression, *GPR*)作为代理模型来预测个体的适应度. 具体而言, 代理模型通过构建基于核函数的非参数化概率模型, 利用历史适应度评估数据对未评估个体的适应度进行预测. 这种方法避免了对每个个体进行昂贵的实际评估, 而是通过代理模型$V'(X)$来估算个体的适应度. 通过结合早期终止机制, 如果个体的预测适应度未达到预定阈值, 则可跳过对该个体的评估, 进一步提高计算效率. ②进一步提出了动态策略调整机制, 可以根据实时情境的反馈优化搜索方向, 确保策略组合符合设备资源约束. 具体而言, 搜索空间根据设备的剩余内存和能耗要求动态缩小, 保留符合当前资源条件的策略. 通过结合跨层优化和情境感知的进化操作, 进化过程中的选择、交叉等操作会根据实时资源约束动态调整, 从而避免不必要的计算和资源浪费, 确保高效且符合实际部署需求. 例如, 为了优化内存占用, 进化过程中的选择和交叉操作会根据内存限制动态调整. 在每一代中, 内存约束函数首先筛选策略, 计算每个策略的内存占用, 若超过内存阈值则排除该策略. 适应度函数通过引入内存惩罚项$\lambda \times M(x_i)$来抑制内存占用较大的策略, 优先选择内存占用较小的个体. 在选择操作中, 采用锦标赛选择机制, 优先挑选内存占用较低的个体. 交叉操作只在内存占用较小的策略之间进行, 避免生成内存过大的后代. 通过这些调整, 进化算法能够动态响应资源约束, 高效地找到符合硬件资源限制的最优策略, 确保在内存受限的环境下实现高效自适应部署.

算法 3. 基于进化算法的多目标优化策略搜索.

输入: 策略空间 E, 种群大小 N, 最大迭代次数 G, 适应度估计$V'(X)$;

输出: 最优策略 (c^*, f^*, r^*).

① $X \leftarrow$ *InitializePopulation* (E, N); /*初始化种群*/

② $D \leftarrow \varnothing$; /*适应度记录初始化*/

③ for each $x \in X$ do

④ $V'(X) \leftarrow$ *EvaluateFitness* (x); /*计算初始种群适应度*/

⑤ *UpdateGPR* $(GPR, x, V'(X))$; /*更新 *GPR* 模型*/

⑥ end for

⑦ for $t = 1$ to G do /*迭代优化*/

⑧ $X' \leftarrow$ *TournamentSelection* $(X, V'(X))$; /*锦标赛选择生成父代*/

⑨ for each $(x_1, x_2) \in X'$ do

⑩ $x_{child} \leftarrow$ *Crossover* (x_1, x_2, p_c); /*执行交叉*/

⑪ $V'(x_{child}) \leftarrow$ *PredictFitness* (GPR, x_{child}); /*使用 *GPR* 预测适应度*/

⑫ *UpdateGPR* $(GPR, x_{child}, V'(x_{child}))$; /*更新 *GPR* 模型*/

⑬ $D \leftarrow D \cup \{(x_{child}, V'(x_{child}))\}$; /*更新适应度记录*/

⑭ end for

⑮ $X \leftarrow$ *Replace* $(X, X', V'(X))$; /*更新种群*/

⑯ end for

⑰ $(c^*, f^*, r^*) \leftarrow$ *BestIndividual* (D); /*提取最优解*/

⑱ return (c^*, f^*, r^*).

4.3 运行时在线性能评估

在性能评估过程中, 我们在目标数据集和移动终端设备上衡量网络结构的表现, 以找到满足多目标约束的最优网络结构. 为避免动态推理中的重训练问题, 我们在离线阶段利用参数共享和权重装载技术训练包含不同变体的压缩算子群C. 在线推理时, 可以灵活选择不同的压缩算子变体c、算子融合机制f和算子重排技术r, 以动态调整模型配置, 提升推理性能.

具体来说, 离线阶段首先通过反向传播训练一个高精度的骨干网络M. 然后, 利用参数转换技术学习 4 种不同的压缩算子及其不同的压缩率, 并存储其额外权重副本. 我们设定了目标精度阈值, 只有在实际精度$Acc(M, c, f, r)$低于该阈值时才会微调压缩算

子变体c的转换参数，因此仅需少量额外参数来存储这些转换参数.

通过在骨干网络M上预训练不同压缩算子变体c的整体性能（例如内存$S(M,c)$、时延$Latency(M,c)$等），按照不同性能指标进行模型结构重要性排序，指导运行时的选择. 在线推理时，基于本节在线性能评估得到的性能配置文件，指导进化算法搜索最优跨层组合策略，从而优化推理性能. 这种方法在离线阶段确保模型精度$Acc(M,c,f,r)$，并在在线推理时灵活调整模型配置a^*，以获得更好的性能而无需重训练.

5 实验验证

本节首先对实验设置进行介绍，其次对基于可伸缩图算结构的跨层优化方法中的压缩算子变体、算子融合机制以及算子重排技术分别进行实验验证，最后在实际部署的运行时情境进行自适应最优策略搜索并对实验结果进行分析.

5.1 实验设置

1）数据集. 本文采用CIFAR-100数据集对各骨干网络以及不同压缩率的不同压缩算子变体进行预训练，并对各层优化方法以及运行时情境自适应策略进行推理验证. 在算子重排实验中，我们采用MNIST[69]和Fashion-MNIST[70]验证模型性能.

2）模型. 在实验过程中，我们使用了5种常见的深度学习模型：ResNet18, VGG16, AlexNet, Inceptionv1, WideResNet28_10. 这些模型在精度和性能开销上有不同的平衡，被广泛应用于移动设备上的深度学习任务.

3）实验设备. 实验设备包括：①边缘设备，即GPU设备Nvidia Jetson Orin Nano（D1）和CPU设备Raspberry Pi 4B（D2）. ②移动设备，即Android设备小米9手机（D3）. 以验证方法在不同架构终端设备上的有效性.

4）基准方法. 本文采用7种基准方法进行对比实验：①使用CIFAR-100数据集预训练的原模型，未进行任何优化操作. ②MobileNetv2[71]. MobileNetv2用具有线性BottleNeck的反向残差代替传统的卷积操作，通过扩展到高维并使用深度卷积进行卷积操作. ③SVD[62]. 基于SVD的卷积分解技术在$conv_i$和$conv_{(i+1)}$之间引入了一个额外的卷积层. 插入层的神经元数量k根据$conv_i$中的动态神经元数量m设置. ④Fire[66]. SqueezeNet中提出的模型压缩方法Fire减少了卷积核大小，并使用Squeeze层减少输入通道数. ⑤AdaDeep[9]. AdaDeep自动选择并组合压缩技术来生成一个专门的DNN，以平衡精度和资源约束. ⑥BiP[72]. BiP通过引入双线性结构优化剪枝-重训练流程，实现基于双层优化（BLO）的剪枝方法. ⑦GhostNetv2[73]. GhostNetv2通过引入DFC注意力机制优化了特征表示，提升了长距离空间信息捕获能力，在保持计算效率的同时实现了精度与推理速度更好的平衡.

5）实际移动场景. 为了验证方法在不同移动场景下的适应性能，我们模拟了4种不同的性能偏好场景. ①精度优先场景（S1）. 在医学影像分析、人脸识别支付等应用中，任务质量更加依赖模型的推理精度. ②内存优先场景（S2）. 在设备并行进程增加时，模型推理进程可用内存更加受限. ③时延优先场景（S3）. 在增强现实（AR）应用中，任务质量更加依赖模型的响应速度和推理时延. ④标准场景（S4）. 标准环境下用户没有显著性能偏好，模型各项性能较平衡.

5.2 可伸缩图算结构的离线性能增益

可伸缩图计算结构在模型层、计算图层和内存调度层分别进行了优化，旨在通过不同的优化方法组合来平衡各种限制条件下的性能需求. 因此，本节将分别开展以下实验：不同压缩算子变体的性能对比、可伸缩模型的算子融合对性能的影响，以及算子重排在内存优化方面的表现. 通过这些实验，为后续的运行时自适应搜索提供了丰富且有效的搜索空间.

5.2.1 不同压缩算子变体的性能对比

为了验证不同压缩算子变体的性能及其相较于原始模型的性能提升，以AlexNet作为骨干模型对各个压缩算子变体进行预训练，并在CIFAR-100数据集上对比精度、时延等性能指标. 压缩算子变体包括不同类型和不同压缩率的压缩算子，具体包括：Bottleneck结构块、SVD结构块、Fire结构块以及DSC. 不同类型的压缩算子具有不同的压缩率，Bottleneck和SVD结构块的压缩率可设置为25%, 50%, 75%；Fire结构块的压缩率可设置为12.5%和25%.

压缩算子变体为轻量化模型结构提供了一种有效的方法，能够在不显著影响精度的前提下，降低时延和参数量. 图6展示了这10种压缩算子变体在时延和参数量方面的对比. 结果表明，与原始模型AlexNet相比，使用压缩算子变体优化后的模型在时延和参数量上都有显著改善，其中时延减少了0.69%~34.5%，参数量减少了21.3%~80.7%. 在同一类型的压缩算子中，较高压缩率的变体表现出更低的时延和更小的参数量. 然而，压缩率越高，精度损失也越大. 如图7所示，除压缩率为12.5%的Fire结构块外，其他压缩算子变体的精度损失均不超过5%，而Fire结构块在12.5%压缩率下的精度损失达到10%.

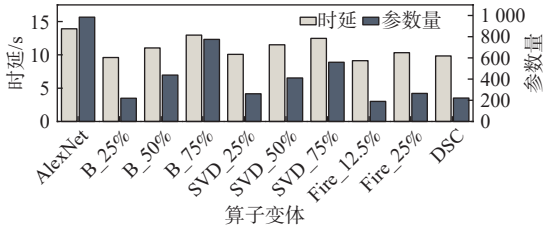


Fig. 6 Model parameter count and runtime latency under different operator variants

图 6 不同算子变体下模型参数量和运行时延

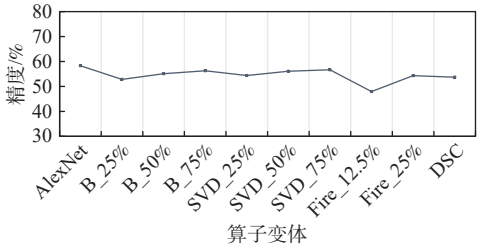


Fig. 7 Comparison of model accuracy under different operator variants

图 7 不同算子变体下模型精度对比

通过减少冗余参数及其计算, 压缩算子变体在显著降低时延的同时, 仅对精度造成了微小的影响. 然而, 过高的压缩率会导致精度的显著下降, 这需要在追求轻量化和维持模型精度之间进行权衡.

5.2.2 可伸缩模型的算子融合性能影响

为了突破传统的固定类型算子融合, 本文提出了一种特定结构的融合方法, 根据算子输入和输出的映射关系分为 One-to-One, Many-to-Many, Reorganize 这 3 种类型. 通过分析预训练模型及其压缩算子变体的结构, 选取了 Many-to-Many 和 One-to-One 类型的算子进行融合, 具体融合规则包括 Conv+BN, Conv+BN+ReLU, Concat+ReLU. 对预训练模型及其变体中所有满足融合规则的算子进行融合, 以减少冗余访存开销, 从而降低时延.

实验结果表明, 算子融合规则应用于不同压缩算子变体上, 都对时延具有显著优化效果, 同时保持其精度不变. 实验分别展示了不同压缩算子类型和压缩率在不同融合规则下的性能表现. 如图 8 所示, 基于 Bottleneck 结构块, 分别在压缩率为 25%, 50%, 75% 的变体上进行不同融合规则的实验验证. 在不同压缩率的变体中, 无论采用何种融合规则, 模型精度始终保持一致. 当压缩率为 25% 时, 不同融合规则带来了 4.7%~35.0% 的时延降低; 压缩率为 50% 时, 时延降低了 7.7%~34.8%; 压缩率为 75% 的变体在融合后, 时延降低了 10.7%~20.4%.

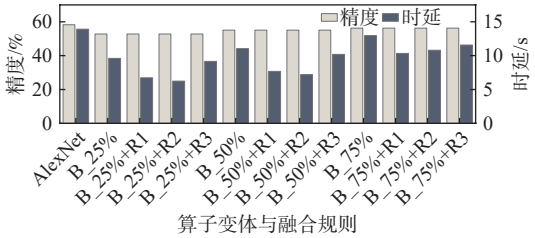


Fig. 8 Operator fusion performance of operator variants with different compression rates

图 8 不同压缩率的算子变体的算子融合性能

图 9 展示了在压缩率固定为 25% 的情况下, 不同类型的压缩算子(Bottleneck 结构块、SVD 结构块、Fire 结构块和深度可分离卷积块)进行算子融合的性能优化情况. 实验结果显示, Bottleneck 结构块的时延降低了 4.7%~35%, SVD 结构块的时延降低了 13.3%~24.8%, Fire 结构块的时延降低了 16.8%~ 25.1%, DSC 的时延降低了 8.7%~ 20.0%.

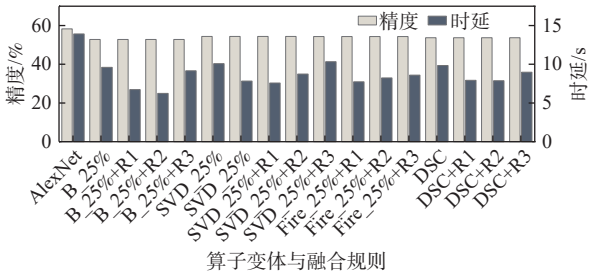


Fig. 9 Operator fusion performance of different types of compression operator variants

图 9 不同类型压缩算子变体的算子融合性能

时延优化效果明显的情况说明此时压缩算子变体的模型结构中包含大量可融合算子, 优化空间较大. 算子融合通过优化存在数据依赖的算子, 减少中间张量数据的访存开销和运算开销, 不改变模型结构, 因此对精度没有影响. 这一优化方法为性能的综合提升做出了重要贡献.

5.2.3 算子重排的内存优化表现

为了验证不同算子重排方法对内存优化的效果, 并探索最优的内存优化算子重排方法, 我们在树莓派上使用 ResNet18 和 Inceptionv1 模型对 MNIST 和 Fashion-MNIST 数据集进行推理实验. 本节将分别对计算图分支的拓扑重排优化、逐元素算子的重排优化以及这 2 种方法的联合优化进行实验验证.

1)计算图分支的拓扑重排优化

计算图分支的拓扑重排的主要目的是减少内存中保存的张量数量, 从而节省内存并优化推理性能. 为了验证方法的有效性, 我们首先对计算图分支部

分进行分析. 以 ResNet18 为例, 图 10 展示了计算图中部分优化的情况. 对优化前后的模型信息进行分析, 结果如表 7 所示. 在 ResNet18 中, 2 个分支经过优化. 我们对比了优化前后的算子执行顺序、内存中保留的张量, 以及内存占用情况(即存储在内存中的张量总和).

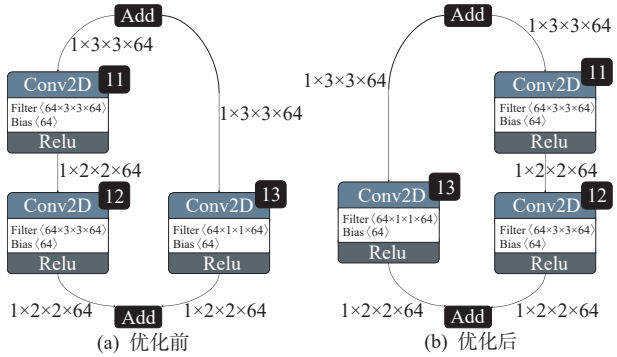


Fig. 10 Illustration of topological reordering optimization of computational graph branches

图 10 计算图分支的拓扑重排优化示意图

Table 7 Performance of Topological Reordering Methods for Computational Graph Branches

表 7 计算图分支的拓扑重排方法性能

分支	优化前			优化后		
	算子执行顺序	内存中保留的张量	内存占用/KB	算子执行顺序	内存中保留的张量	内存占用/KB
分支1	Conv2D_11	[60, 61]	3.328	Conv2D_13	[60, 63]	3.328
	Conv2D_12	[60, 61, 62]	4.352	Conv2D_11	[60, 61, 63]	4.352
	Conv2D_13	[60, 62, 63]	4.352	Conv2D_12	[61, 62, 63]	3.072
分支2	Conv2D_16	[67, 68]	1.280	Conv2D_18	[67, 70]	1.280
	Conv2D_17	[67. 68, 69]	1.536	Conv2D_16	[67, 68, 70]	1.536
	Conv2D_18	[67, 69, 70]	1.536	Conv2D_17	[68, 69, 70]	0.768
总内存占用/KB			16.384	总内存占用/KB		14.336
平均内存占用/KB			2.731	平均内存占用/KB		2.389

以分支 1 为例, 算子执行顺序的改变导致内存中保留的张量发生变化, 从而引起内存占用的波动. 优化后, 内存中需持续存储的张量从 60 号变为 63 号. 如图 11 所示, ResNet18 优化前后的分支 1 对比显示, 63 号张量的内存占用更小, 因此减少了内存消耗. 图 11 中还显示, 61~63 号张量的内存占用基本相同, 而 60 号张量的内存占用最大. 因此, 优化的关键在于避免 60 号张量长时间存储在内存中. 优化前, 60 号张量一直占用内存, 而优化后, 该张量在最后一次卷积计算中被释放, 从而节省了内存. 同理, 分支 2 中, 通过提前释放 67 号张量, 进一步减少了内存使用. 与 ResNet18 的实验类似, 我们对 Inceptionv1 优化前后的模型信息进行了分析, 如图 12 所示. 分支 1 中每个张量所占的内存大小相同, 这表明内存占用的关键在于内存中存储的张量数量. 通过对比图中优化前后内存中张量存储的结果, 可以看出内存的节省是由于存储的张量数量减少.

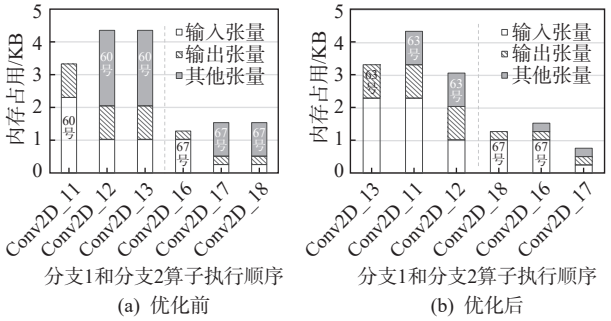


Fig. 11 Illustration of tensor distribution in memory

图 11 内存中张量分布示意图

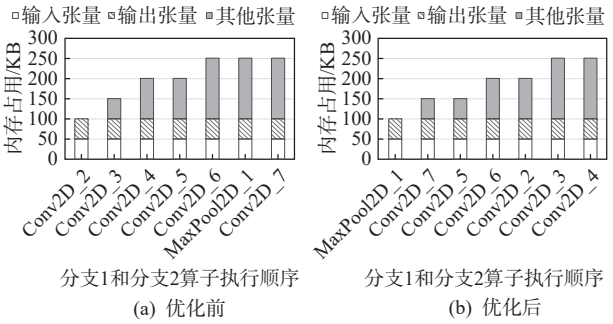


Fig. 12 Illustration of tensor distribution in memory

图 12 内存中张量分布示意图

实验数据表明, 重新拓扑排序可以减少内存中保存的张量数量, 从而达到节省内存的目的, 进而优化推理性能. 值得注意的是, 在整个过程中, 模型结构保持不变, 因此模型的精度未受到任何影响. 拓扑重排在内存优化和性能提升方面具有显著效果, 同时确保了模型的稳定性和准确性.

2)逐元素类算子的重排优化

逐元素类算子的重排优化可以在不影响精度的前提下显著降低内存占用和计算量, 从而提升整体推理性能. 通过对模型进行优化, 图 13 展示了 ResNet18 计算图中的部分优化情况. 优化后, StrideSlice 算子从 Conv-BN-ReLU 算子之后移至 Conv 算子之后, Batch Normalization 和 ReLU 算子之前. 不同于之前的方法, 本实验分析了整个计算图的内存占用和计算量. 首先, 在服务器上进行精度测试, 以确定逐元素类算子重排是否影响精度. 优化前后分别进行了 10 次测试, 结果如图 14 所示, 验证了逐元素类算子重排对精度没有显著影响.

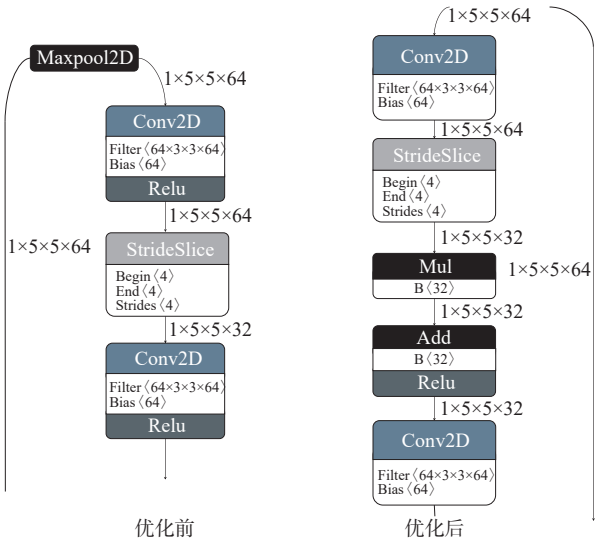


Fig. 13 Illustration of reordering optimization of element-wise operators

图 13 逐元素类算子的重排优化示意图

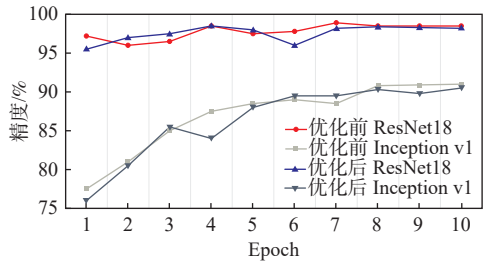


Fig. 14 Accuracy comparison based on forward shift of StrideSlice operator

图 14 基于 StrideSlice 算子前移的精度对比

表 8 的数据显示, 优化后的 ResNet18 和 Inception-v1 模型在内存占用和计算量上都有显著减少. 对于 ResNet18, 经过逐元素类算子的重排后, 内存占用减少了 33.57%, 计算量减少了 28.94%. 在 Inceptionv1 模型中, 内存占用和计算量分别减少了 37.75% 和 5.64%. 这些实验结果验证了逐元素类算子重排方法的有效性. 通过该方法, 不仅显著节省了内存, 还有效降低了计算量, 从而优化了推理性能. 这一优化策略在保持模型精度的同时, 提升了整体性能.

Table 8 Experimental Results of ResNet18 and Inceptionv1 Models

表 8 ResNet18 和 Inceptionv1 模型的实验结果

模型	推理效率	优化前	优化后
ResNet18	内存占用/KB	474.45	315.19
	计算量/MFLOP	4.993	3.548
Inceptionv1	内存占用/B	34.36	21.39
	计算量/MFLOP	87.503	82.566

逐元素类算子的重排通常将算子前移. 对于裁剪类算子, 提前执行可减少冗余数据, 保留重要特征, 降低内存占用和计算量. 对于其他计算类算子, 提前计算可减少数据长久保留, 降低存储开销, 从而减少内存占用. 这些算子通常不影响精度, 因为它们的计算基于对应位置的元素值.

3)分支类和逐元素类重排联合优化

计算图分支的拓扑重排优化与逐元素类算子重排优化是 2 个正交的方法, 联合应用这 2 个方法可实现内存占用的最小化. 如表 9 所示, 本文通过消融实验评估了优化方法的效果. 比较了原始模型、仅应用分支类优化、仅应用逐元素类优化以及联合优化的 4 种情况. 结果表明, 相比于未优化的原始模型, 单独应用分支类优化和逐元素类优化均有效降低了内存占用. 具体而言, ResNet18 的内存占用分别减少了 5.90% 和 33.57% 而 Inceptionv1 的内存占用分别减少了 11.93% 和 37.75%. 联合优化在这 2 个模型上的内存占用分别减少了 35.25% 和 39.41%, 显示了最佳的优化效果.

Table 9 Ablation Experimental Results of Each Optimization Method

表 9 合优化方法的消融实验结果

模型	推理效率	原始模型	分支类	逐元素类	联合优化
ResNet18	内存占用/KB	474.45	446.46	315.19	307.19
	计算量/MFLOP	4.993	4.993	3.548	3.548
Inceptionv1	内存占用/MB	34.36	30.26	21.39	20.82
	计算量/MFLOP	87.503	87.503	83.497	83.497

拓扑重排优化仅改变算子的执行顺序, 不影响模型结构和精度, 因此计算量未减少. 计算量的减少主要来源于逐元素类算子的重排优化. 本节主要关注内存占用的优化效果.

5.3 运行时情境自适应策略搜索性能测试

5.2 节介绍了静态的跨层优化方法, 但这些方法无法适应运行时动态变化的场景. 因此, 本节旨在根据运行时平台的资源限制和用户需求, 探索最优的跨层优化方法组合, 以实现整体性能的最佳配置. 我们将针对不同环境、模型以及设备, 进行运行时情境自适应策略的性能测试, 以评估各策略在实际应用中的效果和适应性.

5.3.1 AdaInfer 的运行时情景自适应能力验证

为了验证 AdaInfer 在具有不同性能偏好的移动场景下的适应能力, 我们将基于 CIFAR-100 预训练的 ResNet18 模型部署在按照 S1→S2→S3→S4 连续变化的动态环境下, 并测试不同方法的推理性能.

AdaInfer 在动态环境下能够迅速适应当前情境，搜索得到当前场景下偏好指标最优的部署方案，并保持其他各项性能指标平衡. 实验结果如表 10 所示，相比于其他优化方法，AdaInfer 在对精度、时延和内存占用分别具有高优先度的 S1, S2, S3 环境下，取得了各自场景下高优先级优化指标的最优结果.

Table 10 Performance Comparison of AdaInfer and Benchmark Methods in Different Environments

表 10 AdaInfer 在不同环境下与基准方法的性能对比

方法	S1			S2			S3			S4		
	精度/%	时延/s	内存占用/MB	精度/%	时延/s	内存占用/MB	精度/%	时延/s	内存占用/MB	精度/%	时延/s	内存占用/MB
原始模型	72.56	465.09	2 937.02	72.56	465.09	2 937.02	72.56	465.09	2 937.02	72.56	465.09	2 937.02
MobileNetv2	57.89	409.57	2 528.80	57.89	409.57	2 528.80	57.89	409.57	2 528.80	57.89	409.57	2 528.80
SVD	71.91	337.39	2 476.02	71.91	337.39	2 476.02	71.91	337.39	2 476.02	71.91	337.39	2 476.02
Fire	69.31	180.72	2 064.33	69.31	180.72	2 064.33	69.31	180.72	2 064.33	69.31	180.72	2 064.33
AdaDeep	68.25	270.36	2 008.79	68.25	270.36	2 008.79	68.25	270.36	2 008.79	68.25	270.36	2 008.79
AdaInfer（本文）	71.91	291.71	1 882.00	62.08	153.33	1 431.16	67.35	135.63	1 825.21	70.93	233.59	1 693.14

在 S1 环境下，AdaInfer 在精度方面相比于 MobileNetv2, Fire, AdaDeep 分别提升了 14.02 个百分点，2.60 个百分点，3.66 个百分点. 为保障在精度优先场景下的模型推理精度，AdaInfer 优先应用不损害精度的优化技术（如算子融合、算子重排），并尽可能提高多变体压缩算子的压缩率以保证精度. 在 S2 环境下，AdaInfer 的内存占用相比于 MobileNetv2, SVD, Fire, AdaDeep 分别降低了 43.41%, 42.20%, 30.67%, 28.76%. 在这一环境下，AdaInfer 优先考虑减少内存占用的优化技术（如算子重排、多变体可伸缩模型），并适当放宽精度和时延的要求. 在 S3 环境下，AdaInfer 的时延相比于 MobileNetv2, SVD, Fire, AdaDeep 分别降低了 66.89%, 59.84%, 24.81%, 50%. 为在时延优先场景下优化模型推理速度，AdaInfer 优先应用优化时延的技术（如算子融合、多变体可伸缩模型），同时放宽对精度和内存占用的要求.

在没有高优先级指标的标准场景 S4 下，AdaInfer 通过综合运用跨层优化技术实现了精度与运行开销的最佳平衡. 在此环境下，AdaInfer 在精度、内存占用和时延方面的表现至少优于 MobileNetv2, SVD, Fire, AdaDeep 中的 2 个性能指标. 无论 S4 后的场景偏好如何，AdaInfer 均能迅速适应并调整优化策略，以确保性能的持续优越及其运行时的自适应能力.

然而，其他基准方法法缺乏运行时情境自适应能力，无法在动态变化的移动场景中提供有效的优化. 一些基准方法采用一次性优化策略，未能针对环境变化进行调整，导致其在不同性能需求下的表现不如 AdaInfer. 因此，这些方法在面对动态环境时往往难以实现最佳性能，显示出其在应对移动场景动态变化中的局限性.

5.3.2 AdaInfer 的模型通用性验证

为了验证 AdaInfer 的模型通用性，我们在 Jetson Orin Nano 上部署了 ResNet18, VGG16, WideResNet28_10 这 3 种不同的模型，并在 CIFAR-100 数据集上进行了实验测试.

综合精度、内存占用和时延 3 个性能指标的对比，AdaInfer 在具有不同架构的模型上表现出最优性能，能够在保证精度的前提下显著降低时延和内存开销，证明了 AdaInfer 的模型通用性. 如图 15 所示，AdaInfer 在所有测试模型中，精度损失始终保持在 5% 以内，并且在与 MobileNetv2, SVD, Fire, AdaDeep 等基准方法对比时，AdaInfer 的精度始终高于至少 2 个基准方法.

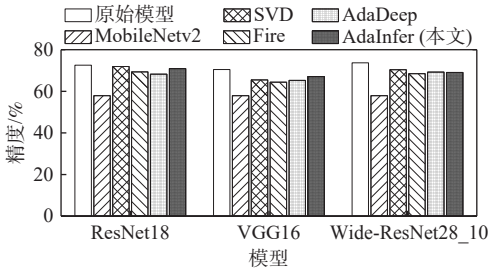


Fig. 15 Accuracy comparison of AdaInfer on different models

图 15 AdaInfer 在不同模型上的精度对比

图 16 展示了在不同模型上，AdaInfer 相较于原始模型及其他基准方法，在内存性能上均表现出显著改进. 具体而言，AdaInfer 在 3 个模型上的内存占用降低了 12.80%~42.35%，相较于原始模型，SVD, Fire, AdaDeep 均有显著优化. 根据图 17 的结果，AdaInfer 在所有测试模型中的时延表现也优于至少 3 个基准

方法, 平均时延降低幅度为 13.79%~73.89%.

5.3.3 AdaInfer 的设备通用性验证

为了验证 AdaInfer 在不同设备上的通用性, 我们分别在边缘设备 Jetson Orin Nano(D1)和 Raspberry Pi 4B(D2)以及移动设备小米 9 手机(D3)上部署了预训练的 ResNet18 模型, 并使用 CIFAR-100 数据集进行实验测试. 在不同设备上, AdaInfer 展现了最优的运行时情境适应能力, 能够有效应对平台资源的变化并及时调整策略以适应当前的计算资源. 因此, AdaInfer 具有良好的设备通用性, 能够在多样化的硬件环境中保持最优性能.

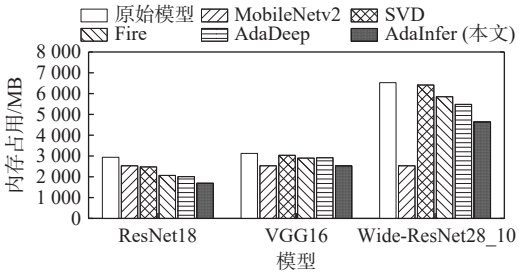


Fig. 16 Comparison of memory use of AdaInfer on different models

图 16 AdaInfer 在不同模型上的内存占用对比

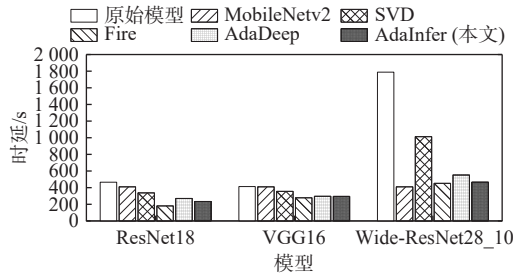


Fig. 17 Comparison of latency of AdaInfer on different models

图 17 AdaInfer 在不同模型上的时延对比

1)边缘设备上 AdaInfer 的性能

如表 11 所示, 尽管基准方法的精度和参数量在 D1 和 D2 上保持不变(这是由预训练模型的结构决定的), 由于设备的后端和底层系统差异, 两者之间的时延和内存开销存在显著差异. AdaInfer 能够根据设备平台和计算资源的变化调整优化策略, 以实现性能和开销的最佳平衡. 实验数据表明, D1 的计算能力优于 D2, 并且在同一设备上, AdaInfer 相比每个基准方法至少在 2 个性能指标上具有优势.

2)移动设备上 AdaInfer 的性能

我们在基于 Android 系统的移动设备 D3 上, 评估了各基准方法在 VGG16 网络架构和 CIFAR-100 数据集上的推理性能. 如图 18 所示, AdaInfer 在移动设备上实现了最佳的精度-性能平衡, 在推理精度、时

Table 11 Performance Comparison of AdaInfer and Benchmark Methods on Edge Devices

表 11 AdaInfer 在边缘设备下与基准方法的性能对比

方法	D1				D2			
	精度/%	内存占用/MB	时延/s	参数量	精度/%	内存占用/MB	时延/s	参数量
原始模型	72.56	1 300.80	4.65	11.22×10^6	72.56	912.77	8.48	11.22×10^6
MobileNetv2	57.89	1 110.34	4.10	2.37×10^6	57.89	509.14	8.10	2.37×10^6
SVD	71.91	1 149.37	3.37	5.63×10^6	71.91	601.71	7.17	5.63×10^6
Fire	69.31	1 151.66	1.81	2.23×10^6	69.31	846.25	5.88	2.23×10^6
AdaDeep	68.25	1 121.18	2.70	0.90×10^6	68.25	846.23	6.58	0.90×10^6
AdaInfer(本文)	70.93	1 061.16	2.34	3.90×10^6	69.15	550.98	5.08	2.19×10^6

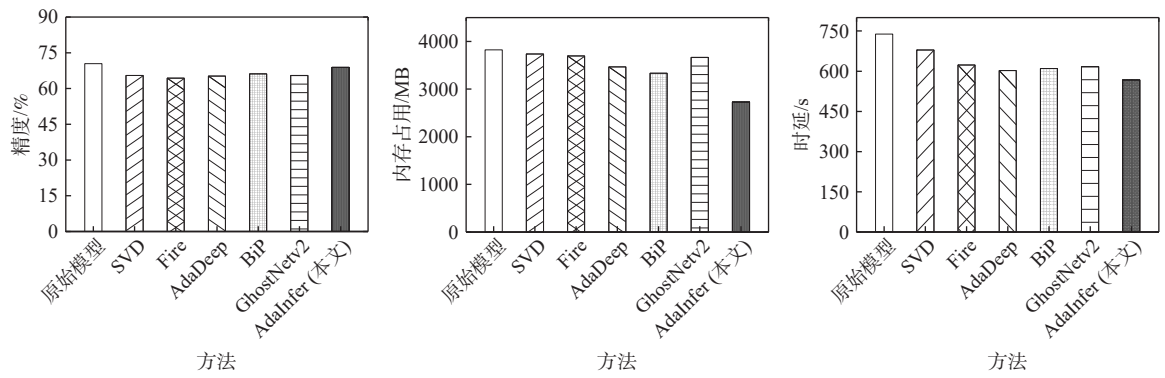


Fig. 18 Performance comparison of AdaInfer and benchmark methods on mobile device

图 18 AdaInfer 在移动设备下与基准方法的性能对比

延和内存占用方面均优于 BiP 和 GhostNetv2. 具体而言, AdaInfer 在推理精度上达到了 68.91%, 比 BiP 和 GhostNetv2 分别提高了 2.74 个百分点和 3.41 个百分点; 在内存占用方面, AdaInfer 分别比 BiP 和 GhostNetv2 减少了 18.19% 和 25.61%; 在推理时延上, AdaInfer 比 BiP 和 GhostNetv2 分别减少了 42.95 s 和 49.63 s. AdaInfer 的精度优势来源于动态自适应特性, 能够在可用资源波动时动态调整跨层优化方案以充分利用硬件资源, 在满足约束的条件下实现更高的精度. 此外, AdaInfer 通过算法与编译层的跨层优化, 扩展了优化空间和优化策略选择, 既突破了单层优化方法的性能上限, 又避免了传统模型压缩方法过度牺牲精度来达到优化目标.

5.4 运行时开销

为了量化情境变化带来的额外感知开销, 本研究将 AdaInfer 部署在基于 Jetson 的智能小车上进行图像分类任务, 并根据不同情境的需求进行性能优化. 当光照条件较差或小车发生颠簸时, 需要优先提升模型精度, 以处理低质量图像, 如昏暗或模糊的场景(S1); 当多个进程并发运行导致内存资源紧张时, 需优化内存占用以确保任务稳定执行(S2); 而当小车行驶速度超过 2 m/s 时, 由于场景变化迅速, 需要提升推理响应速度, 以确保任务的实时性(S3). 为了有效监控情境变化, 本实验设置了 1 个 10 s 的感知时间窗口, 每个时间窗口内包括 3 个感知步骤: 首先, 通过相机采集图像并提取光照信息, 时延开销为 33.33 ms、内存开销为 4 505.6 KB、能耗开销为 50 039.82 μJ, 主要用于图像处理和光照特征提取; 其次, 使用惯性测量单元(IMU)监测小车的振动情况及行驶速度, 时延开销为 1.01 ms、内存开销为 0.011 7 KB、能耗开销为 2.64 μJ, 主要用于动态调整推理任务的响应能力; 最后, 通过系统监控接口获取当前可用内存, 时延开销为 0.98 ms、内存开销为 0.012 3 KB、能耗开销为 2 439.17 μJ, 旨在确保设备资源紧张时及时调整策略以适应内存缩减.

综合以上开销, 感知步骤的总时延开销为 35.32 ms, 占据 10 s 时间窗口的 3.532‰, 总内存开销为 4 505.624 KB, 占该任务内存总开销的 4.139‰, 总能耗开销为 52 481.63 μJ, 提供了一个量化的指标, 以评估在不同情境下通过感知反馈调整任务执行时的资源开销.

6 总 结

本文解决了在动态部署环境下持续运行的移动应用程序中, DNN 的运行时自适应算法-系统跨层优化问题. 我们提出了 AdaInfer, 这是一种环境自适应且在运行时进行跨层优化的 DNN 框架. AdaInfer 通过快速调整和组合模型层、计算图层和内存层的优化策略, 持续平衡多个性能指标. 我们设计了一种与模型结构无关的可伸缩图算结构, 并基于此提出了模型层、计算图层和内存层的跨层优化方法, 能够根据不同设备的硬件资源和平台自动调整, 以最大化部署效率. 我们将跨层优化策略的运行时调整问题建模为动态优化问题, 通过一组随时变化的约束条件来模拟动态环境. 为此, 我们提出了一种高效的搜索策略, 包括跨层精简搜索空间构建、多目标跨层优化策略搜索和运行时在线性能评估, 以提升本地在线搜索的效率和质量. 在实验中, 我们在 3 种常见的移动和边缘设备上测试了不同移动应用程序的 5 种模型, 并在多种持续变化的移动场景中进行了自适应搜索部署方案. 实验结果展示了 AdaInfer 在动态运行环境中的显著优势. 在未来工作中, 面对多样化和动态的移动场景(如数据、任务和平台), 特别是在不断变化的硬件资源和用户需求条件下, 进一步提升 DNN 优化的实时性和自适应性, 将成为改进该工作的有效方向, 这将实现更高效、更智能的移动计算解决方案.

作者贡献声明: 罗诗妍负责论文总体框架及方法设计和实验验证; 刘思聪负责研究问题及研究意义的确立、方法设计的指导以及论文修改; 郭斌负责对论文内容的整体把控和修改优化; 方程负责相关工作、方法设计和研究挑战部分的写作与修改; 王敏帆负责部分实验验证和格式修改; 郭赛负责部分实验验证; 於志文负责对全文内容的整体把控.

参 考 文 献

[1] Li Xiangyang, Shang Fei, Yan Yubo, et al. Survey on low power sensing of AIoT[J]. Journal of Computer Research and Development, 2024, 61(11): 2754−2775 (in Chinese)
(李向阳, 尚飞, 闫宇博, 等. 智能物联网低功耗感知综述[J]. 计算机研究与发展, 2024, 61(11): 2754−2775)

[2] Yang Zhican, Yu Chun, Zheng Fengshi, et al. Proxitalk: Activate speech input by bringing smartphone to the mouth[J]. Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies, 2019, 3(3): 1−25

[3] Kwon H, Tong C, Haresamudram H, et al. IMUTube: Automatic extraction of virtual on-body accelerometry from video for human activity recognition[J]. Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies, 2020, 4(3): 1−29

[4] Wang Xiaofei, Han Yiwen, Leung V, et al. Convergence of edge computing and deep learning: A comprehensive survey[J]. IEEE Communications Surveys & Tutorials, 2020, 22(2): 869−904

[5] Luo Jianhao, Wu Jianxin. AutoPruner: An end-to-end trainable filter pruning method for efficient deep model inference[J]. Pattern Recognition, 2020, 107: 107461

[6] Swaminathan S, Garg D, Kannan R, et al. Sparse low rank factorization for deep neural network compression[J]. Neurocomputing, 2020, 398: 185−196

[7] Narayanan D, Harlap A, Phanishayee A, et al. PipeDream: Generalized pipeline parallelism for DNN training[C]//Proc of the 27th ACM Symp on Operating Systems Principles. New York: ACM, 2019: 1−15

[8] He Yihui, Lin Ji, Liu Zhijian, et al. AMC: Automl for model compression and acceleration on mobile devices[C]//Proc of the European Conf on Computer Vision(ECCV). Berlin: Springer, 2018: 784−800

[9] Liu Sicong, Du Junzhao, Nan Kaiming, et al. AdaDeep: A usage-driven, automated deep model compression framework for enabling ubiquitous intelligent mobiles[J]. IEEE Transactions on Mobile Computing, 2020, 20(12): 3282−3297

[10] Yang Yu, Liu Di, Fang Hui, et al. Once for all skip: Efficient adaptive deep neural networks[C]//Proc of the 26th Design, Automation & Test in Europe Conf & Exhibition (DATE). Piscataway, NJ: IEEE, 2022: 568−571

[11] Li Gen, Yun I, Kim J, et al. Dabnet: Depth-wise asymmetric bottleneck for real-time semantic segmentation[J]. arXiv preprint, arXiv: 1907.11357, 2019

[12] Liu Sicong, Guo Bin, Fang Cheng, et al. Enabling resource-efficient AIoT system with cross-level optimization: A survey[J]. IEEE Communications Surveys & Tutorials, 2023, 26(1): 389−427

[13] Liu Sicong, Guo Bin, Ma Ke, et al. AdaSpring: Context-adaptive and runtime-evolutionary deep model compression for mobile applications[J]. Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies, 2021, 5(1): 1−22

[14] Chen Xing, Li Ming, Zhong Hao, et al. DNNOff: Offloading DNN-based intelligent IoT applications in mobile edge computing[J]. IEEE Transactions on Industrial Informatics, 2021, 18(4): 2820−2829

[15] Xu Zichuan, Zhao Liqian, Liang Weifa, et al. Energy-aware inference offloading for DNN-driven applications in mobile edge clouds[J]. IEEE Transactions on Parallel and Distributed Systems, 2020, 32(4): 799−814

[16] He Yihui, Zhang Xiangyu, Sun Jian. Channel pruning for accelerating very deep neural networks[C]//Proc of the 16th IEEE Int Conf on Computer Vision. Piscataway, NJ: IEEE, 2017: 1389−1397

[17] Liberis E, Lane N. Differentiable neural network pruning to enable smart applications on microcontrollers[J]. Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies, 2023, 6(4): 1−19

[18] Zhou Yiren, Moosavi-Dezfooli S, Cheung N, et al. Adaptive quantization for deep neural network[C]//Proc of the 32nd AAAI Conf on Artificial Intelligence. Palo Alto, CA: AAAI, 2018: 4596−4604

[19] Jin Haoyu, Wu Donglei, Zhang Shuyu, et al. Design of a quantization-based dnn delta compression framework for model snapshots and federated learning[J]. IEEE Transactions on Parallel and Distributed Systems, 2023, 34(3): 923−937

[20] Sainath T, Kingsbury B, Sindhwani V, et al. Low-rank matrix factorization for deep neural network training with high-dimensional output targets[C]//Proc of the 38th IEEE Int Conf on Acoustics, Speech and Signal Processing. Piscataway, NJ: IEEE, 2013: 6655−6659

[21] Cai Gaoyuan, Li Juhu, Liu Xuanxin, et al. Learning and compressing: Low-rank matrix factorization for deep neural network compression[J]. Applied Sciences, 2023, 13(4): 2704

[22] Chen Guobin, Choi W, Yu Xiang, et al. Learning efficient object detection models with knowledge distillation[C/OL]//Proc of the 31st Conf on Neural Information Processing Systems. New York: Curran Associates, 2018[2024-06-20]. https://proceedings.neurips.cc/paper/2017/hash/e1e32e235eee1f970470a3a6658dfdd5-Abstract.html

[23] Mishra R, Gupta H. Designing and training of lightweight neural networks on edge devices using early halting in knowledge distillation[J]. IEEE Transactions on Mobile Computing, 2023, 23(5): 4665−4677

[24] Shi Ruiwen, Li Guanghui, Dai Chenglong, et al. Feature-oriented and decoupled network structure based filter pruning method[J]. Journal of Computer Research and Development, 2024, 61(7): 1836−1849 (in Chinese)
(施瑞文, 李光辉, 代成龙, 等. 一种基于特征导向解耦网络结构的滤波器修剪方法[J]. 计算机研究与发展, 2024, 61(7): 1836−1849)

[25] Basha S, Farazuddin M, Pulabaigari V, et al. Deep model compression based on the training history[J]. Neurocomputing, 2024, 573: 127257

[26] Jiang Xiaotang, Wang Huan, Chen Yiliu, et al. MNN: A universal and efficient inference engine[J]. Proceedings of Machine Learning and Systems, 2020, 2: 1−13

[27] Chen Tianqi, Moreau T, Jiang Ziheng, et al. {TVM}: An automated {End-to-End} optimizing compiler for deep learning[C]//Proc of the 13th USENIX Symp on Operating Systems Design and Implementation (OSDI'18). Berkeley, CA: USENIX Association, 2018: 578−594

[28] Roesch J, Lyubomirsky S, Kirisame M, et al. Relay: A high-level compiler for deep learning[J]. arXiv preprint, arXiv: 1904.08368, 2019

[29] Paszke A, Gross S, Massa F, et al. Pytorch: An imperative style, high-performance deep learning library[C/OL]//Proc of the 33rd Conf on Neural Information Processing Systems. New York: Curran Associates, 2019[2024-06-20]. https://proceedings.neurips.cc/paper/2019/hash/bdbca288fee7f92f2bfa9f7012727740-Abstract.html

[30] Jia Zhihao, Thomas J, Warszawski T, et al. Optimizing DNN computation with relaxed graph substitutions[J]. Proceedings of Machine Learning and Systems, 2019, 1: 27−39

[31] Zhou Yanqi, Roy S, Abdolrashidi A, et al. Transferable graph optimizers for ml compilers[C]//Proc of the 34th Conf on Neural Information Processing Systems. New York: Curran Associates, 2020: 13844−13855

[32] Mirhoseini A, Goldie A, Pham H, et al. A hierarchical model for

device placement[C]//Proc of the 6th Int Conf on Learning Representations. Piscataway, NJ: IEEE, 2018: 1−12

[33] Gao Yuanxiang, Chen Li, Li Baochun. Spotlight: Optimizing device placement for training deep neural networks[C]//Proc of the 35th Int Conf on Machine Learning. New York: ACM, 2018: 1676−1684

[34] Addanki R, Venkatakrishnan S, Gupta S, et al. Placeto: Learning generalizable device placement algorithms for distributed machine learning[J]. arXiv preprint, arXiv: 1906.08879, 2019

[35] Zheng Zhen, Zhao Pengzhan, Long Guoping, et al. FusionStitching: Boosting memory intensive computations for deep learning workloads[J]. arXiv preprint, arXiv: 2009.10924, 2020

[36] Hu Shimin, Liang Dun, Yang Guoye, et al. Jittor: A novel deep learning framework with meta-operators and unified graph execution[J]. Science China Information Sciences, 2020, 63: 1−21

[37] Ivanov A, Dryden N, Ben-Nun T, et al. Data movement is all you need: A case study on optimizing transformers[J]. Proceedings of Machine Learning and Systems, 2021, 3: 711−732

[38] Niu Wei, Guan Jiexiong, Wang Yanzhi, et al. DNNFusion: Accelerating deep neural networks execution with advanced operator fusion[C]//Proc of the 42nd ACM SIGPLAN Int Conf on Programming Language Design and Implementation. New York: ACM, 2021: 883−898

[39] Zheng Zhen, Yang Xuanda, Zhao Pengzhan, et al. AStitch: Enabling a new multi-dimensional optimization space for memory-intensive ML training and inference on modern SIMT architectures[C]//Proc of the 27th ACM Int Conf on Architectural Support for Programming Languages and Operating Systems. New York: ACM, 2022: 359−373

[40] Xie Kunpeng, Yi Dezhi, Liu Yiqing, et al. SAF-CNN: A sparse acceleration framework of convolutional neural network for embedded FPGAs[J]. Journal of Computer Research and Development, 2023, 60(5): 1053−1072 (in Chinese)
(谢坤鹏, 仪德智, 刘义情, 等. SAF-CNN: 面向嵌入式 FPGA 的卷积神经网络稀疏化加速框架[J]. 计算机研究与发展, 2023, 60(5): 1053−1072)

[41] Mu Yudong, Li Wenming, Fan Zhihua, et al. Data flow architecture optimization for YOLO neural networks[J/OL]. Chinese Journal of Computers, [2024-11-27]. http://kns.cnki.net/kcms/detail/11.1826.TP.20240620. 1532.002.html (in Chinese)
(穆宇栋, 李文明, 范志华, 等. 面向 YOLO 神经网络的数据流架构优化研究[J/OL]. 计算机学报, [2024-11-27]. http://kns.cnki.net/kcms/detail/11.1826.TP.20240620.1532.002.html)

[42] Rhu M, Gimelshein N, Clemons J, et al. vDNN: Virtualized deep neural networks for scalable, memory-efficient neural network design[C]//Proc of the 49th Annual IEEE/ACM Int Symp on Microarchitecture (MICRO). Piscataway, NJ: IEEE, 2016: 1−13

[43] Ren Jie, Luo Jiaolin, Wu Kai, et al. Sentinel: Efficient tensor migration and allocation on heterogeneous memory systems for deep learning[C]//Proc of the 27th IEEE Int Symp on High-Performance Computer Architecture (HPCA). Piscataway, NJ: IEEE, 2021: 598−611

[44] Jung J, Kim J, Lee J. DeepUM: Tensor migration and prefetching in unified memory[C]//Proc of the 28th ACM Int Conf on Architectural Support for Programming Languages and Operating Systems. New York: ACM, 2023: 207−221

[45] Wang Zirui, Jiang Dejun. Key techniques of swapping mechanism based on ultra-Low latency SSD[J]. Journal of Computer Research and Development, 2024, 61(3): 557−570 (in Chinese)
(王紫芮, 蒋德钧. 基于超低延迟 SSD 的页交换机制关键技术[J]. 计算机研究与发展, 2024, 61(3): 557−570)

[46] Moon J, Kapoor P, Lee J, et al. NNTrainer: Light-weight on-device training framework[J]. arXiv preprint, arXiv: 2206.04688, 2022

[47] Wang Qipeng, Xu Mengwei, Jin Chao, et al. Melon: Breaking the memory wall for resource-efficient on-device machine learning[C]// Proc of the 20th Annual Int Conf on Mobile Systems, Applications and Services. New York: ACM, 2022: 450−463

[48] Zhang Chuanqi, Wang Sa, Sun Ninghui, et al. GroupUCP: An on-demand fine-grained cache dynamic partition strategy[J/OL]. Journal of Computer Research and Development, [2024-11-27]. https://crad.ict.ac.cn/cn/article/doi/10.7544/issn1000-1239.202440061(in Chinese)
(张传奇, 王卅, 孙凝晖, 等. GroupUCP: 按需动态调节的细粒度 Cache 划分策略[J]. 计算机研究与发展, [2024-11-27]. https://crad.ict.ac.cn/cn/article/doi/ 10.7544/issn1000-1239.202440061)

[49] Chen Tianqi, Xu Bing, Zhang Chiyuan, et al. Training deep nets with sublinear memory cost[J]. arXiv preprint, arXiv: 1604.06174, 2016

[50] Zong Zan, Lin Li, Lin Leilei, et al. STR: Hybrid tensor re-generation to break memory wall for DNN training[J]. IEEE Transactions on Parallel and Distributed Systems, 2023, 34(8): 2403−2418

[51] Liberis E, Lane N. Neural networks on microcontrollers: Saving memory at inference via operator reordering[J]. arXiv preprint, arXiv: 1910.05110, 2019

[52] Lin Ji, Zhu Ligeng, Chen Weiming, et al. On-device training under 256KB memory[C]//Proc of the 36th Conf on Neural Information Processing Systems. New York: Curran Associates, 2022: 22941−22954

[53] Steiner B, Elhoushi M, Kahn J, et al. MODeL: Memory optimizations for deep learning[C]//Proc of the 40th Int Conf on Machine Learning. New York: ACM, 2023: 32618−32632

[54] Han S, Shen Haichen, Philipose M, et al. Mcdnn: An approximation-based execution framework for deep stream processing under resource constraints[C]//Proc of the 14th Annual Int Conf on Mobile Systems, Applications, and Services. New York: ACM, 2016: 123−136

[55] Yang Taojiannan, Zhu Sijie, Chen Chen, et al. Mutualnet: Adaptive convnet via mutual learning from network width and resolution[C]// Proc of the 16th European Conf on Computer Vision (ECCV). Berlin: Spinger, 2020: 299−315

[56] Marco V, Taylor B, Wang Z, et al. Optimizing deep learning inference on embedded systems through adaptive model selection[J]. ACM Transactions on Embedded Computing Systems, 2020, 19(1): 1−28

[57] Wu Zuxuan, Nagarajan T, Kumar A, et al. Blockdrop: Dynamic inference paths in residual networks[C]//Proc of the 31st IEEE Conf on Computer Vision and Pattern Recognition. Piscataway, NJ: IEEE, 2018: 8817−8826

[58] Teerapittayanon S, McDanel B, Kung H. Branchynet: Fast inference via early exiting from deep neural networks[C]//Proc of the 23rd Int Conf on Pattern Recognition (ICPR). Piscataway, NJ: IEEE, 2016:

2464−2469

[59] Gao Xitong, Zhao Yiren, Dudziak Ł, et al. Dynamic channel pruning: Feature boosting and suppression[J]. arXiv preprint, arXiv: 1810.05331, 2018

[60] Yang Yu, Liu Di, Fang Hui, et al. Once for all skip: Efficient adaptive deep neural networks[C]//Proc of the 25th Design, Automation & Test in Europe Conf & Exhibition (DATE). Piscataway, NJ: IEEE, 2022: 568−571

[61] Sun Zhensu, Du Xiaoning, Song Fu, et al. When neural code completion models size up the situation: Attaining cheaper and faster completion through dynamic model inference[C]//Proc of the 46th IEEE/ACM Int Conf on Software Engineering. New York: ACM, 2024: 1−12

[62] Lane N, Bhattacharya S, Georgiev P, et al. Deepx: A software accelerator for low-power deep learning inference on mobile devices[C]//Proc of the 15th ACM/IEEE Int Conf on Information Processing in Sensor Networks (IPSN). Piscataway, NJ: IEEE, 2016: 1−12

[63] Liu Sicong, Lin Yingyan, Zhou Zimu, et al. On-demand deep model compression for mobile devices: A usage-driven model selection framework[C]//Proc of the 16th annual Int Conf on Mobile Systems, Applications, and Services. New York: ACM, 2018: 389−400

[64] Szegedy C, Vanhoucke V, Ioffe S, et al. Rethinking the inception architecture for computer vision[C]//Proc of the 29th IEEE Conf on Computer Vision and Pattern Recognition. Piscataway, NJ: IEEE, 2016: 2818−2826

[65] He Kaiming, Zhang Xiangyu, Ren Shaoqing, et al. Deep residual learning for image recognition[C]//Proc of the 29th IEEE Conf on Computer Vision and Pattern Recognition. Piscataway, NJ: IEEE, 2016: 770−778

[66] Iandola F, Han Song, Moskewicz M, et al. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and< 0.5 MB model size[J]. arXiv preprint, arXiv: 1602.07360, 2016

[67] Howard A, Zhu Menglong, Chen Bo, et al. MobileNets: Efficient convolutional neural networks for mobile vision applications[J]. arXiv preprint, arXiv: 1704.04861, 2017

[68] Jia Fucheng, Zhang Deyu, Cao Ting, et al. CoDL: Efficient CPU-GPU co-execution for deep learning inference on mobile devices[C]//Proc of the 20th annual Int Conf on Mobile Systems, Applications, and Services. New York: ACM, 2022, 22: 209−221

[69] LeCun Y, Bottou L, Bengio Y, et al. Gradient-based learning applied to document recognition[J]. Proceedings of the IEEE, 1998, 86(11): 2278−2324

[70] Xiao Hao, Rasul K, Vollgraf R. Fashion-MNIST: A novel image dataset for benchmarking machine learning algorithms[J]. arXiv preprint, arXiv: 1708.07747, 2017

[71] Sandler M, Howard A, Zhu M, et al. MobileNetv2: Inverted residuals and linear bottlenecks[C]//Proc of the 31st IEEE Conf on Computer Vision and Pattern Recognition. Piscataway, NJ: IEEE, 2018: 4510−4520

[72] Zhang Yihua, Yao Yuguang, Ram P, et al. Advancing model pruning via bi-level optimization[C]//Proc of the 35th Conf on Neural Information Processing Systems. New York: Curran Associates, 2022: 18309−18326

[73] Tang Yehui, Han Kai, Guo Jianyuan, et al. GhostNetv2: Enhance cheap operation with long-range attention[C]//Proc of the 35th Conf on Neural Information Processing Systems. New York: Curran Associates, 2022: 9969−9982



Luo Shiyan, born in 2002. Master. Her main research interests include AIoT system and mobile computing.

罗诗妍, 2002 年生. 硕士. 主要研究方向为人工智能物联网系统、移动计算.



Liu Sicong, born in 1992. PhD, associate professor. Her main research interests include AIoT system and mobile computing.

刘思聪, 1992 年生. 博士, 副教授. 主要研究方向为人工智能物联网系统、移动计算.



Guo Bin, born in 1980. PhD, professor. His main research interests include ubiquitous computing, crowdsensing, mobile crowdsensing.

郭 斌, 1980 年生. 博士, 教授. 主要研究方向为普适计算、群智感知、移动群智感知.



Fang Cheng, born in 2000. Master. His main research interests include AIoT system and mobile computing.

方 程, 2000 年生. 硕士. 主要研究方向为人工智能物联网系统、移动计算.



Wang Minfan, born in 2002. Master. Her main research interests include AIoT system and mobile computing.

王敏帆, 2002 年生. 硕士. 主要研究方向为人工智能物联网系统、移动计算.



Guo Sai, born in 2001. Master. His main research interest includes computer vison.

郭 赛, 2001 年生. 硕士. 主要研究方向为计算机视觉.



Yu Zhiwen, born in 1977. PhD, professor. His main research interests include ubiquitous computing and AIoT system.

於志文, 1977 年生. 博士, 教授. 主要研究方向为普适计算、人工智能物联网系统.