

7

神经网络架构搜索

7.1 概述

近些年来，随着深度学习技术的发展，神经网络已然成为实现人工智能的一大途径，而神经网络本身的性能和效率又与它们的网络架构密不可分。早些年间，神经网络架构通常在反复试错的手工设计过程中完成，而设计高性能且高效率的神经网络架构通常依赖大量的专家经验和人力资源，这就导致基于手工设计神经网络的周期较长。而在最近这些年，为了解放专家资源，减少不必要的科研投入，研究人员开始逐渐关注于如何应用自动化的算法进行神经网络架构的设计，在满足硬件效率限制的前提下，设计出性能尽可能高的架构。

从优化问题定义和求解的视角来看，神经网络架构自动化设计的优化问题的通用定义如下：**优化空间**主要包括神经网络架构，通常来说，优化空间可以被进一步分解成若干种不同的决策，例如架构的层数、每一层使用的卷积核大小、不同层之间的连接关系等等，优化空间可以被表示为这些决策集合的笛卡尔积。有关优化空间和决策的详细介绍，请读者移步本书 7.3 节；**优化目标**是尽量提升设计的网络架构在特定任务上的性能；**限制条件**可以是硬件指标，例如模型的运行时延、功耗等等，也可以是硬件相关的代理指标，例如模型的参数量、计算量等等。该优化问题可以用公式 7.1 表示，其中 α 表示网络架构， S 表示优化空间， w 表示架构的权重， R_{val} 表示模型的任务性能， L_{train} 表示模型在训练数据集上的损失函数， F 表示硬件相关指标的获取函数， B 表示对应的硬件限制。由于优化目标函数通常黑盒的，不存在显式的函数解析式，因此通常考虑使用搜索的方式求解该黑盒优化问题，神经网络架构搜索 (Neural Architecture Search, NAS) 方法应运而生。

$$\begin{aligned}
 & \underset{\alpha \in S}{\text{maximize}} \quad R_{val}(w^*(\alpha), \alpha) \\
 \text{s.t.} \quad & w^*(\alpha) = \underset{w}{\operatorname{argmin}} L_{train}(w, \alpha) \\
 & F(\alpha) \leq B
 \end{aligned} \tag{7.1}$$

神经网络架构搜索(以下简称 NAS)方法通常包含三个组成部分，分别是搜索空间、搜索策略和评估策略。具体来说，**搜索空间**定义了一个包含所有待探索架构的集合，**搜索策略**给出了在搜索空间中做探索的方式，而**评估策略**则被用来估计被探索到的架构的性能(例如精度、时延等)。NAS方法的常用流程如图7.1所示，控制器基于搜索策略在搜索空间中做采样，将采样得到的架构送入评估器，评估器按照评估策略返回架构的评估性能，控制器利用该评估结果对调整搜索方向，重复迭代上述过程直到获得符合要求的架构或达到预设的迭代次数。根据搜索空间、搜索策略和评估策略的不同，NAS方法的流程会存在一定差异，笔者将在接下来的部分详细介绍它们。

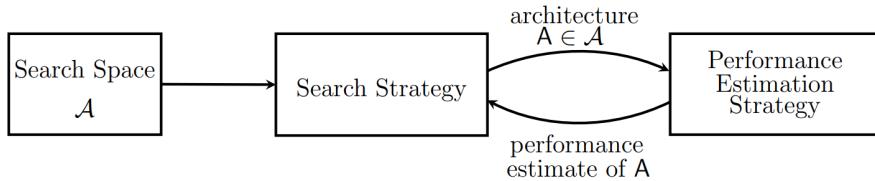


图 7.1 NAS 方法的基本流程^[72]

在特定场景中应用 NAS 方法时，通常需要完成以下四个步骤(如图7.2所示)：

- 设计 (Design)**: 针对给定的任务、硬件和其他的约束条件(例如搜索过程的开销预算、最终架构的资源预算等)，设计合适的搜索空间、搜索策略和评估策略。
- 搜索 (Search)**: 运行 NAS 搜索过程。通常来说，该步骤是最耗时的一步，因此有许多研究工作都关注于此。为了提升 NAS 方法的高效性，使用者可以考虑设计更紧致的搜索空间、提升搜索策略的采样效率或使用速度更快的代理评估策略。
- 架构产生 (Derive)**: 产生最终搜索到的架构。不同的搜索策略往往有不同的架构产生方式，例如，通过性能指标挑选架构^[259]，通过一定的离散化规则决定架构^[196]，或是从控制器网络输出的架构分布中采样架构^[392, 393]。对于大部分基于离散搜索的 NAS 方法来说，产生的最终架构由搜索算法本身定义好的指标所决定，该过程简单直接；但对于可微分 NAS 方法来说，架构在连续空间中的表征和离散空间中的存在差异，因此有专门的研究工作优化该过程^[307]。
- 再训练 (Retrain)**: 对产生的最终架构在目标任务上做一遍单独的训练。通常来

说，为了减少搜索过程的时间开销，NAS 方法不会将探索过程中的候选架构做充分的训练，而是利用不完全准确的代理评估策略做性能估计，例如使用单次评估策略^[246] 或零次评估策略^[3]。因此，最终得到的架构会在任务上重新做训练，得到用于部署的模型。

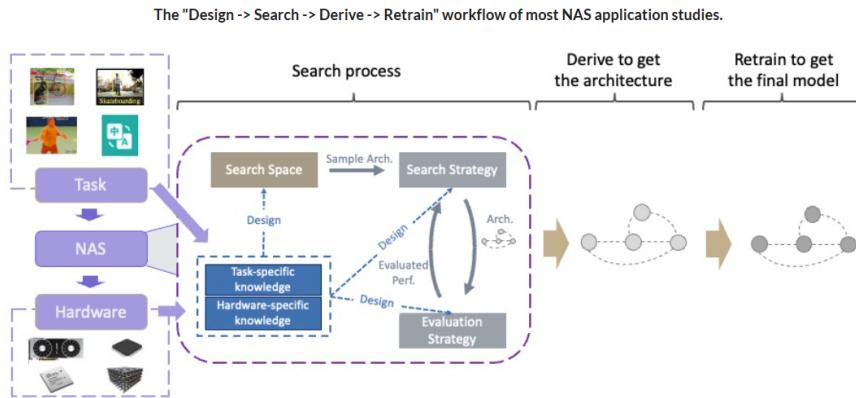


图 7.2 应用 NAS 方法的四个步骤

本章将按以下顺序展开，7.2节将 NAS 领域的发展脉络和现状总结，7.3~7.5节将详细介绍 NAS 方法的三个组件，介绍内容包括对应领域的研究现状和发展趋势，为了使读者能更清晰地理解各组件的工作原理，笔者将对应地选取若干篇具有代表性工作展开介绍。7.6和7.7节将整体介绍两种常用的 NAS 方法，分别是高效的可微分 NAS 和考虑模型复杂度或效率的 NAS。最后，在7.8节笔者将基于一款可扩展继承的 NAS 框架介绍若干个实践案例，帮助感兴趣的读者使用 NAS 方法或做进一步的研究开发。

7.2 领域发展与总结

自 Zoph 等人于 ICLR2017 中发表了 NAS 领域的开山之作 NASRL^[392] 后，大量的研究工作在 NAS 领域涌现。笔者阅读分析了大量研究文献后，总结出了四个 NAS 领域的细分研究方向，如图7.3所示。

NAS 流程加速。为了加速 NAS 流程，较早期的研究设计了高效单次评估策略 (one-shot evaluation strategy)^[16, 196, 246]，并应用不同的搜索策略，包括基于强化学习的搜索算法^[392, 393]、进化算法^[259]、基于梯度的搜索算法^[196]、基于预测器的搜索算法^[215] 等等。沿着这些研究思路，最近的研究工作可以按照优化的 NAS 组件划分为两支：(1) 设计更先进的性能预测器构建方式或训练方式^[238, 266, 315]，提升搜索策略的采样效率。本章将

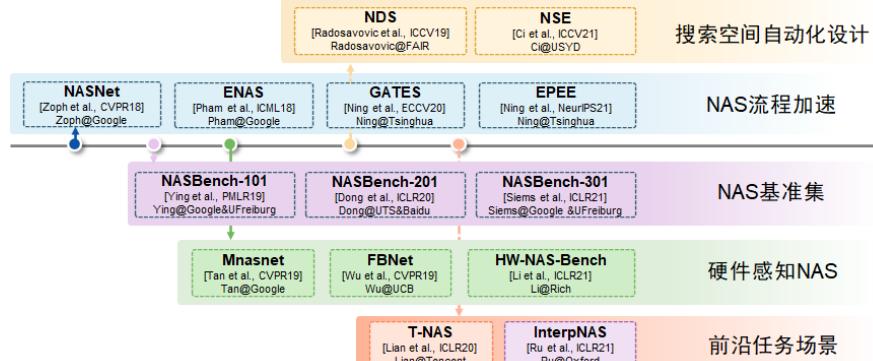


图 7.3 NAS 方法细分领域的时间轴图

在7.4节详细介绍这类基于性能预测器的 NAS 搜索策略; (2) 改进单次评估策略^[284, 385, 390]或设计速度更快的零次评估策略^[3, 187, 222]。本章将在7.5节展开讨论单次评估策略和零次评估策略是什么、性能存在怎样的问题、产生问题的原因以及改进的方向和具体工作。

NAS 基准集。 NAS 领域涌现出许多优秀的研究工作，这些工作都从实验上证明了 NAS 方法的有效性。然而，这些工作的实验配置大多存在差异，主要表现在所使用的架构搜索空间、架构再训练的参数等方面。为了让不同的算法能公平地比较性能，同时增强算法的可复现性，各种各样的 NAS 基准^[67, 257, 274, 354]集被提出。Ying 等人构建了第一个基准集 NAS-Bench-101^[354]，该基准集提供了一个微观架构搜索空间中共 4.23×10^5 个架构的真实性能。随后，Dong 等人构建了一个小规模的基准集 NAS-Bench-201^[67]，其中包含了另一个微观架构搜索空间中的 15625 个架构的性能。考虑到上述两个 NAS 基准集所使用的搜索空间的规模比常用的空间的规模小得多，Siems 等人在一个包含约 10^{18} 个架构的微观架构搜索空间中构建了 NAS-Bench-301^[274]，他们实际训练了 60000 个架构，并利用这些真实性能构建了一个性能预测器来预测空间中其他架构的性能。不同于上述基准集均基于微观架构搜索空间构建，Radosavovic 等人在非拓扑架构搜索空间 ResNet 和 ResNeXt 中构建了新的基准集 NDS^[257]，其中，每个搜索空间中包含 5000 个架构的真实性能。

硬件感知 NAS。 从 2018 年期硬件感知 NAS 吸引了众多研究者的关注，不少研究工作^[30, 296]涌现出来。到了 2019 年末，研究者开始关注并设计“算法-编译-硬件”联合设计方法^[190, 345]来自动化地在更大的跨层次搜索空间中探索设计方案。本章将在7.7节展开介绍几种常用的硬件感知 NAS 方法。

搜索空间自动化设计。 搜索空间的设计对 NAS 方法找到性能优秀的架构来说直观重要。在早期的 NAS 工作中，研究者将专家设计架构的知识和经验融入到搜索空

间的设计中^[296, 393]。一些研究工作^[303, 340]说明了搜索空间的设计很大程度上影响或制约 NAS 方法的作用，因此，验证、探索和分析现有搜索空间设计上的优劣势^[256, 303, 340]，适当减少一些设计先验的约束^[325]，并自动化地提取一些低维且可解释的设计规则^[257]，是目前值得研究的方向。FAIR 团队的研究者于 2019 年在该研究方向上主导了一些有趣的研究，从那之后，一些相关的研究主要关注于搜索空间的自动化缩减^[38, 57, 125]。本章将在 7.3 节介绍搜索空间自动化设计的工作。

前沿任务场景。从 NAS 的问题设定来看，架构的评估有很多耦合因素，比如对数据的需求、对标签的需求和具体的应用任务等。因此，相比于传统的充分监督下单一任务的场景，研究者开始研究在更前沿的任务场景，或约束条件更多的问题设定下应用 NAS 方法，例如数据量更少^[178]、标签量更少^[192, 339, 375] 或跨任务架构搜索^[182] 等等。

7.3 搜索空间

搜索空间是 NAS 方法的重要组成部分之一，它定义了 NAS 的搜索范围，更准确地说，搜索空间是一个包含所有待探索的候选神经网络架构集合。对于 NAS 方法来说，搜索空间的优劣会显著地影响其搜索过程的效率和所搜索到的架构的性能。一方面，搜索过程的时间开销与搜索空间的规模呈正相关，越紧致的搜索空间所需要的搜索轮次往往就越少；另一方面，搜索空间限定了架构搜索的范围，在一定程度上决定了搜索结果的精度上限，规模越大的搜索空间往往包含性能更高的架构。因此，从搜索空间的角度考虑，**搜索效率和搜索精度之间往往存在一定的权衡**：规模较小的搜索空间更容易探索，但会约束 NAS 的搜索潜力；而规模较大的搜索空间能有效提升 NAS 的精度，但会增加搜索过程的时间开销。

搜索空间可以表示为若干种搜索决策集合的笛卡尔积。搜索决策 (decision) 决定了搜索空间中架构的组成性能，每种决策又包含若干个选择 (choice)，对每种决策确定了它的选择后，就能得到一个唯一且确定的神经网络架构。以一个小规模的架构搜索空间为例，如图 7.4 所示，该搜索空间中的网络架构有 3 个卷积层组成，其中，搜索决策分为拓扑决策和非拓扑决策。拓扑决策为第一层和最后一层之间的跳连接是否存在 (记作 d_0)，该决策的选择包括存在或不存在 (即 $d_0 = \{0, 1\}$)；非拓扑决策包括每一层卷积的通道数 (第 i 层记作 d_{i1}) 和卷积核尺寸 (第 i 层记作 d_{i2})，其中，通道数可选 32、64 或 128 (即 $d_{i1} = \{32, 64, 128\}$)，卷积核尺寸可选 1、3 或 5 (即 $d_{i2} = \{1, 3, 5\}$)。综上，该搜索空间 S 可以表示为 $S = d_0 \times \prod_{i=1}^3 d_{i1} \times d_{i2}$ ，代入计算可得 $|S| = 1458$ 。

目前对搜索空间的相关研究主要分为人工设计搜索空间和自动设计搜索空间。人工设计搜索空间，即专家融合早期人工设计架构 (例如 ResNet^[2]，DenseNet^[127] 等) 的经验对搜索空间进行设计。笔者将人工设计搜索空间按其包含的搜索决策类型划分为

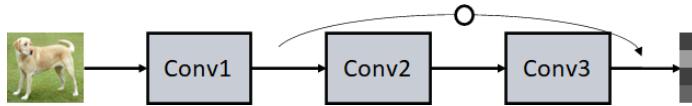


图 7.4 一个小规模的架构搜索空间例子

四类，分别是宏观 (Macro) 架构搜索空间^[392]、微观 (Micro) 架构搜索空间^[393]、非拓扑 (Non-topological)^[295, 319] 架构搜索空间和层次化 (Hierarchical) 架构搜索空间^[194]。人工设计搜索空间是 NAS 领域最早出现的搜索空间类型，其对应的研究较为完备，在主流的 NAS 方法中仍然占据主导地位。自动设计搜索空间，即算法基于对搜索决策的分析和子搜索空间的性能评估，自动化地设计出紧致且高性能的搜索空间。目前该研究方向的代表工作是 Radosavovic 等人^[256] 提出的基于误差分布的搜索空间分析方法，以及半自动化设计的非拓扑搜索空间 RegNet^[257]。从目前的工作来看，对搜索空间的自动化设计仍然处在半自动化阶段，设计过程需要专家的经验指导，例如，RegNet 的产生经过了人工设计和自动化验证这两个步骤的多次交替，即先由研究者提出搜索空间的缩减规则，再自动化的做子空间的性能验证。从该角度看，搜索空间设计的自动化程度仍然有更进一步的可能。

本节将围绕上述两大类研究工作进行展开：7.3.1节将详细介绍三种具有代表性的
人工设计搜索空间，即宏观架构搜索空间、微观架构搜索空间、非拓扑搜索空间和层
次化搜索空间，并简要分析它们的优劣势和适配的应用场景；7.3.2将介绍搜索空间评
估和自动化设计的研究方向，其中重点对 Radosavovic 等人的工作展开讨论。

7.3.1 人工设计搜索空间

宏观架构搜索空间

宏观 (Macro) 架构搜索空间是最早出现的一类搜索空间，其设计在一定程度上沿用了早期的手工设计神经网络的模式。人工设计的卷积神经网络通常包含多个卷积层，且具有一定的拓扑结构。因此，宏观架构搜索空间的搜索决策包括每一层的算子参数和层与层之间的拓扑连接关系。这类搜索空间的代表是 Zoph 等人于 ICLR2017 中提出的工作 NASRL^[392]，如图7.5所示。具体来说，NASRL 搜索空间的搜索决策包括架构每一层 (layer) 中卷积核的数量、尺寸和步长，同时还包括任意两层之间是否有跳连接的存在。这样的决策设计使得该搜索空间能涵盖多样化的架构，包括优秀的手工设计架构，例如 ResNet、DenseNet 等。

宏观架构搜索空间的设计融入了部分人工设计架构的先进经验，具有很高的性

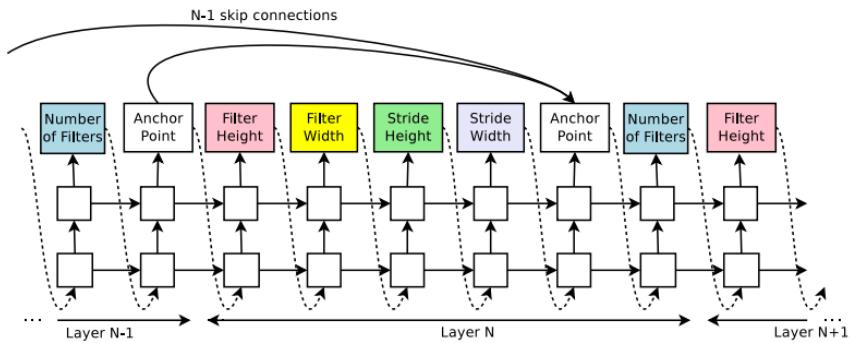


图 7.5 NASRL 使用的宏观架构搜索空间^[392]

能潜力，但该类搜索空间规模过大，导致 NAS 方法的搜索速度非常慢。从相关工作的结果来看，与当时性能最优秀的人工设计神经网络相比，在宏观架构搜索空间中搜索到的网络架构具有同样优秀的性能，例如，在 2016 年，CIFAR-10 数据集上最先进 (State-Of-The-Art, SOTA) 的手工设计网络 DenseNet 获得了 96.54% 的准确率，而 NASRL 设计出的同等规模的网络模型获得了 96.35% 的准确率。然而，NASRL 搜索空间非常庞大，包含了 3.9×10^{73} 个架构。一方面是因为搜索空间规模和架构的层数呈指数级关系，而神经网络的层数通常在几十到上百层，例如 NASRL 的网络架构就有 20 层；另一方面，网络层与层之间的拓扑连接方式非常多，例如 NASRL 中连接方式的数量和架构层数的平方呈指数级关系。庞大的搜索空间规模会导致 NAS 方法耗费大量的时间才能找到性能最好的架构，NASRL 在 CIFAR-10 数据集上训练了 12800 个架构后才找到了与 SOTA 性能相当的网络架构，总搜索开销达到 22400 个 GPU 小时。这样巨大的时间开销难以使 NAS 方法取得广泛应用。

微观架构搜索空间

微观 (Micro) 架构搜索空间的出现很大程度上缓解了宏观搜索空间规模过大的问题，专家借鉴了人工设计神经网络中堆叠可重复结构单元的设计模式，设计了一个仅包含重复的单元架构的空间，即微观架构搜索空间。微观架构搜索空间的搜索决策仅包括单个单元架构中算子的种类和算子之间的连接方式，决策的数量远小于宏观架构搜索空间中的决策数量，因此这类搜索空间的规模相对较小。

微观架构搜索空间的代表工作是 Zoph 等人于 CVPR2018 中发表的 NASNet^[393]，下面将以该工作为例展开介绍微观架构搜索空间的构成。如图7.7所示，完整的神经网络架构由两种不同结构的单元架构通过按照一定的排布方式堆叠而成，这两类单元

架构分别被称为正常 (normal) 单元和降采样 (reduction) 单元。单元架构的堆叠方式一般由人为定义，如图7.6所示，降采样单元将整个神经网络划分为若干个阶段 (stage)，每个阶段均包含 N 个正常单元，借鉴 ResNet 的跳连接结构，每个单元将排在前两个位置的单元的输出作为输入。需要强调的是，正常单元均采用相同单元架构，降采样单元也采用相同单元架构，这样的设计保证了搜索空间的规模不受网络层数的影响。单元架构的堆叠配置与具体的任务或数据集规模有关，例如在 NASNet 中，针对 CIFAR-10 数据集使用 $N = 4$ ，针对 ImageNet 数据集使用 $N = 6$ 。单元架构定义了网络模型的最小可重复单元，通常采用有向无环图 (Directed Acyclic Graph, DAG) 的表示方式，图中每个结点 (node) 代表一张特征图，每条边 (edge) 代表一种计算操作 (例如卷积、池化等)。图7.7展示了一个单元架构的例子，该单元包含 4 个结点，其中 1 号和 2 号结点为输入结点，它们对应的特征图来自前两个单元架构的输出，2 号结点的特征图分别经过 5×5 卷积操作和跳连接操作后，在 3 号结点处聚合成一张新的特征图，接着，1 号结点和 3 号结点的特征图分别经过 3×3 平均池化操作和 3×3 卷积操作后，在 4 号结点处聚合成新的特征图，4 号结点对应的特征图作为单元架构的输出。

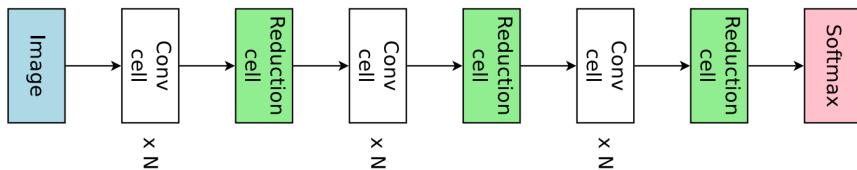
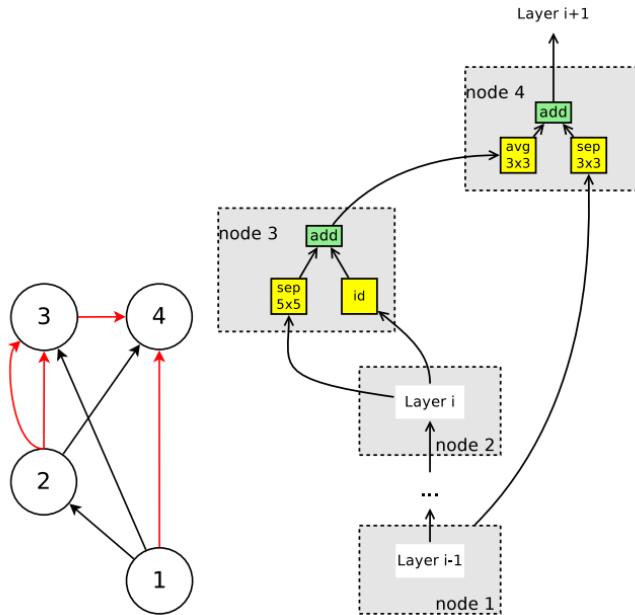


图 7.6 NASNet 中单元架构的堆叠方式^[393]

微观架构搜索空间帮助 NAS 方法在显著减小搜索开销的同时，保持了较高的任务性能。一方面，其借鉴手工神经网络模型由可重复单元架构组成的设计模式，仅包含微型的单元架构，使本身的规模与网络层数无关，显著减小的搜索空间大小；另一方面，其在单元架构中引入了更灵活的连接方式，增加了网络拓扑结构的复杂性和多样性，使搜索空间能涵盖丰富的神经网络结构。例如，NASNet 在 CIFAR-10 数据集上花费 2000 个 GPU 小时，搜索到了准确率达 97.03% 的架构，相比于使用宏观搜索空间的 NASRL 提升了 7 倍的搜索效率。

微观架构搜索空间规模较小且性能优秀，是目前最常用的拓扑搜索空间。然而，微观架构搜索空间由于人为固定了堆叠方式，且每一层的单元架构完全相同，因此在一定程度上牺牲了整体网络结构的灵活性，以及架构的层间多样性。此外，单元架构的过于复杂的拓扑结构也使得该搜索空间中的架构在硬件上的推理延迟较大，难以满足移动端部署的需求。

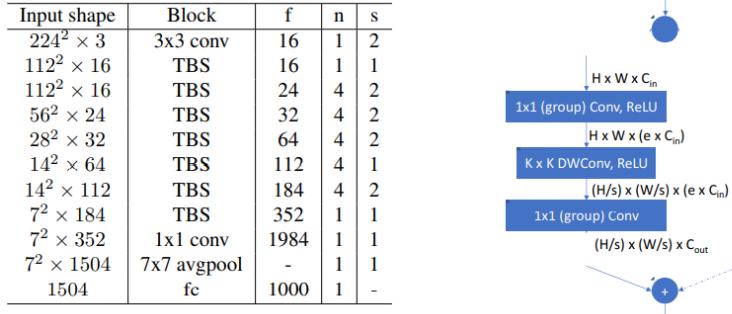
图 7.7 NASNet 单元架构的设计^[393]

非拓扑架构搜索空间

非拓扑 (Non-topological) 架构搜索空间的出现主要是为了提升神经网络模型的硬件效率。虽然微观架构搜索空间显著减小的搜索开销，但一方面，单元架构的拓扑结构过于复杂，在硬件上的推理速度较慢，另一方面，每层相同的结构对模型的精度和时延产生一定的负面影响。为了提升硬件效率和进一步提升模型精度，专家借鉴了轻量级神经网络架构的设计模式，设计了非拓扑架构搜索空间。非拓扑架构搜索空间的搜索决策主要为网络每层选择的模块，以及模块的参数（例如宽度、瓶颈系数等），其中模块通常为人工设计的轻量级模块，相关介绍见??。

非拓扑架构搜索空间的代表工作是 Wu 等人于 CVPR2019 中发表的 FBNet^[320]。如图7.8所示，图左侧为 FBNet 搜索空间的宏观架构，由人为规定，其中“TBS”表示待搜索的轻量级模块。图右侧为轻量级模块的模板，其中分组卷积的组数、深度可分离卷积的尺寸和模块的扩张系数为搜索决策。需要注意的是，该搜索空间中每层轻量级模块的参数可以不同，这样的设计增加了神经网络模型的层间多样性 (Layer Diversity)，实验^[295, 320]证明，允许不同层具有不同的结构能提升网络模型的精度和硬件效率。

非拓扑架构搜索空间能帮助 NAS 方法搜索到精度和效率同时优秀的网络架构。

图 7.8 FBNet 使用的非拓扑架构搜索空间^[320]

例如，在 2019 年手工设计的轻量化神经网络 ShuffleNet，在 ImageNet 数据集上达到了 74.9% 的精度和 33.3 ms 的推理延时，是当时的 SOTA 性能。而 FBNet 设计出的网络架构同样获得了 74.9% 的精度，但推理延时仅为 28.1ms。相比于微观架构搜索空间，非拓扑架构搜索空间的优势主要有以下三点：(1) 摈弃了复杂的拓扑结构，转而使用简单的串行结构，使得模型更适合硬件端的部署。(2) 大量使用深度可分离卷积代替传统卷积，减少了运算开销。(3) 不同的块可以有不同的结构，显著提升了层间多样性，实验证明这样的设计更有利于架构的性能和效率。凭借低延迟、高精度的网络模型，非拓扑架构搜索空间成为了工业界更常用的搜索空间，特别在针对移动端设计轻量级架构时，非拓扑架构搜索空间占据主导地位。

层次化架构搜索空间

层次化 (Hierarchical) 架构搜索空间沿用了微观架构搜索空间的设计思路，但不同的是，在这类搜索空间中，单元架构的组成是层次化的，即高层次 (**high-level**) 的架构由低层次的架构 (**low-level**) 组合而成。层次化架构搜索空间的代表工作是 Liu 等人于 ICLR2018 年中发表的 HierNAS^[194]。如图7.9所示，该图展示了一个三层的单元架构的层次化结构，第一层 (最底层) 的架构是单个运算单元，如图中所示的 $o_1^{(1)}$ (1 × 1 卷积层)、 $o_2^{(1)}$ (3 × 3 卷积层) 和 $o_3^{(1)}$ (3 × 3 最大池化层)。第二层的架构由第一层的架构组合而成，如图中的 $o_1^{(2)}$ 、 $o_2^{(2)}$ 和 $o_3^{(2)}$ 所示。第三层 (最高层) 的架构是以第二层的架构为基本组成单元进行构建，如图中的 $o_1^{(3)}$ 所示。对于每一层构建的架构 (如图中的 $G_1^{(2)}$ 和 $G_1^{(3)}$)，层次化架构搜索空间沿用了微观架构搜索空间中有向无环图的表示形式，这样的设计使神经网络的拓扑结构更加多样。在层次化架构搜索空间中，单元架构的堆叠方式如图7.10所示，与微观架构搜索空间不同的是，每一个阶段 (stage) 只包含一个

正常单元，同时降采样单元直接用步长为 2 的卷积层代替，不再需要进行搜索。

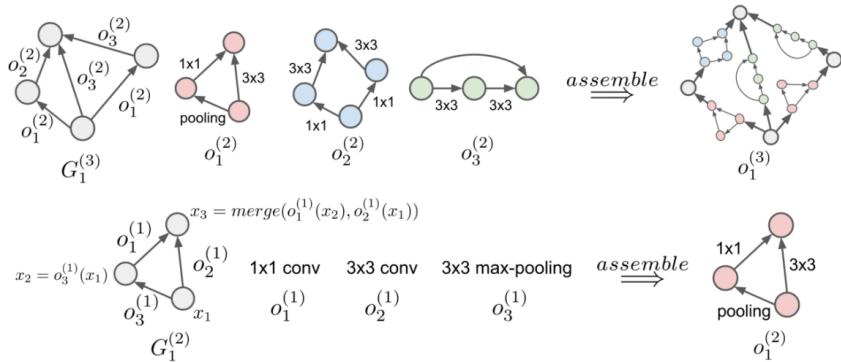


图 7.9 层次化架构搜索空间中单元架构的组成^[194]

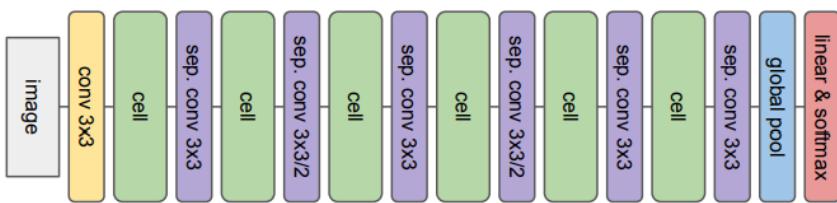


图 7.10 层次化架构搜索空间中单元架构的堆叠方式^[194]

层次化架构搜索空间在神经网络模型的构建上加入了更多的灵活性，从堆叠重复单元架构的角度来看，层次化架构搜索空间用更灵活的层次化设计代替了人工固定的堆叠方式，一定程度上增加了搜索空间的多样性和复杂性。然而，层次化架构搜索空间在引入灵活性的同时，也增大了搜索空间的规模，导致搜索开销增加。此外，层次化架构搜索空间中的架构具有更加复杂的拓扑结构，其硬件推理效率较差。

7.3.2 自动设计搜索空间

人工设计搜索空间，尤其是微观架构搜索空间和非拓扑搜索空间，在神经网络架构搜索任务中获得了显著的效果。然而，这类基于人工经验设计的搜索空间仍然存在一定的缺陷，包括：(1) 空间规模依然庞大，例如目前最常用的一种微观架构搜索空间 DARTS，仍然包含约 10^{18} 种网络架构；(2) 探索到的架构难以具备优秀的可解释性，同时其设计过程也难以带来新的知识。然而，要设计出大小紧致且具备可解释性的优

秀搜索空间不容易，这需要研究者有充分的神经网络架构设计经验，同时也需要反复的试错和调优，这就导致了**搜索空间的人工设计周期长**。例如，目前最常用的微观架构搜索空间仍然是 Zoph 等人在 2018 年提出的 NASNet。

为了解决上述问题，有研究者提出将自动化的思想同样应用到**搜索空间的设计中去**，实现**搜索空间的自动化设计，缩短设计周期**。自动设计搜索空间是 NAS 领域一个较为前沿的子领域，该子领域的代表工作是由 Radosavovic 等人提出的一系列方法，包括设计对搜索空间进行评估的方法^[256]，并基于该方法进行半自动化的搜索空间设计，成功设计出了一个紧致且质量更高的非拓扑搜索空间^[257]。本节将以 Radosavovic 等人的工作为例，介绍自动设计搜索空间的先进方法和研究成果。

搜索空间评估

传统的搜索空间比较和评估大多利用**点估计 (Point Estimation)** 法，即用**搜索空间中任务精度最高的架构的性能作为该搜索空间的评价指标**。但这种评估方式存在两方面的缺陷：(1) 搜索空间的大小可能会误导评价结果，例如从同一个搜索空间种采样两个大小不同的子空间 A 和 B ($|A| > |B|$)，在大多数情况下点估计法会认为子空间 A 具有更好的设计准则，但实际上两个子空间来自同一个搜索空间，遵循完全相同的设计方式。(2) 架构的复杂度可能会误导评价结果，例如从同一个搜索空间的不同复杂度区域采样两个子空间 C 和 D ，其中 C 中包含复杂度较大的网络架构，由于一般情况下，复杂度大的网络具有更高的性能，因此点估计法会偏向于子空间 C ，但同样的，两个子空间的设计差异并不大。

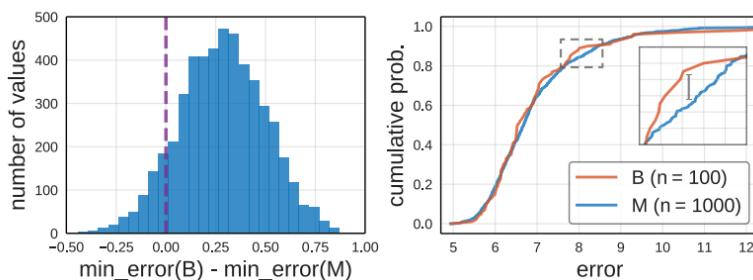


图 7.11 点估计法和误差分布函数法的比较， B 和 M 为大小不同的子空间^[256]

为了解决上述的问题，Radosavovic 等人设计了一种基于采样的**误差经验分布函数 (Empirical Distribution Function, EDF)** 来评估搜索空间的性能。具体来说，给定一个大小为 n 的模型集合，模型的误差集合为 $\{e_i\}$ ，则误差经验分布函数为：

$$F(e) = \frac{1}{n} \sum_{i=1}^n \mathbf{1}[e_i < e] \quad (7.2)$$

$F(e)$ 实际表示的是误差小于 e 的模型比例。使用该分布函数绘制出的曲线进行搜索空间的评估和比较能有效解决子空间大小带来的误导现象，如图7.11所示，左侧为点估计法的结果，可以看到两个空间被认为具有较大差异，而右侧两个子空间的累计分布曲线几乎重合，说明两个子空间具有相似的性能分布。此外，为解决模型复杂度带来的误导问题，Radosavovic 等人设计了一种**正则化误差经验分布函数**。具体来说，对于一个大小为 n 的模型集合，误差集合为 $\{e_i\}$ ，复杂度集合为 $\{c_i\}$ 。定义正则化系数集合为 $\{w_i\}$ ，满足 $\sum_i w_i = 1$ 。定义正则化复杂度经验分布函数为：

$$C(c) = \sum_{i=1}^n w_i \mathbf{1}[c_i < c] \quad (7.3)$$

定义正则化误差经验分布函数为：

$$\hat{F}(e) = \sum_{i=1}^n w_i \mathbf{1}[e_i < e] \quad (7.4)$$

对于带比较的两个搜索空间，找到两组正则化系数使得对于任意 c 满足 $C_1(c) \approx C_2(c)$ ，此时比较 $\hat{F}_1(e)$ 和 $\hat{F}_2(e)$ ，就能消除模型复杂度对评估结果的误导，如图7.12所示。

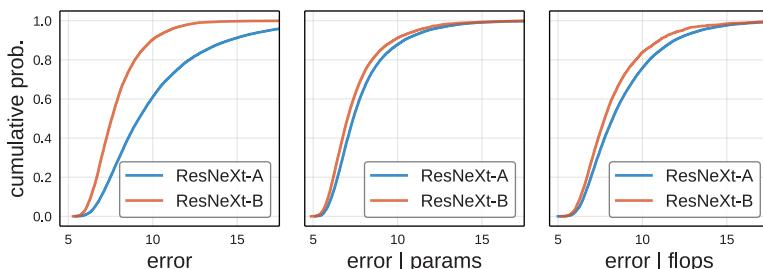


图 7.12 点估计法，误差分布函数法和正则化误差分布函数法的比较，ResNeXt-A 和 ResNeXt-B 从同一个搜索空间的不同复杂度区域采样得到^[256]

搜索空间(半)自动化设计

借助上述基于误差经验分布函数的搜索空间评估方法，Radosavovic 等人^[257] 进一步设计了一种半自动化的设计算法，成功设计了一个紧致且质量更好的搜索空间

RegNet。RegNet 的设计思想如图7.13所示，通过在较大的搜索空间(如图中的A)中加入一些认为的限制条件，逐步获得较小规模的子空间(如图中的B和C)，利用子空间的分布来评价其性能，保证在子空间的整体质量更好。

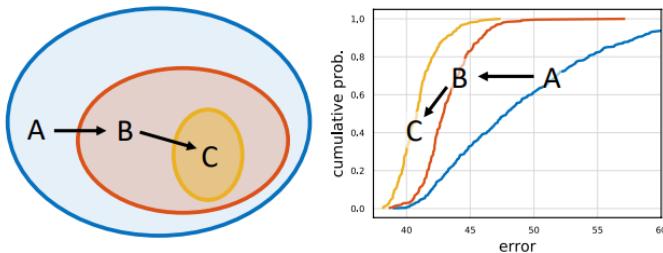


图 7.13 RegNet 的基本设计思路^[257]

RegNet 搜索空间参照非拓扑搜索空间的设计模式。如图7.14所示，整体网络由茎(stem)、躯干(body)和头部(head)组成，其中待搜索的参数集中在躯干部分。网络的躯干包括若干个阶段(stage)，每个阶段内部包括若干个块(block)，需要注意的是，同一个阶段中的块完全相同(除了第一个块的步长与其他块不同)。块的设计借鉴了ResNet中残差瓶颈(residual bottleneck)的设计方式，如图7.15(a)所示，X block由2个1x1的普通卷积和1个3x3的分组卷积组成，两个1x1卷积用来控制通道宽度。第*i*层X block的待搜索参数包括宽度*w_i*，瓶颈系数*b_i*和分组卷积的组数*g_i*。

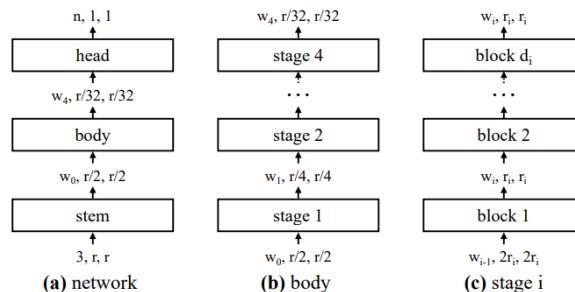


图 7.14 RegNet 搜索空间的结构^[257]

RegNet 的具体设计过程如图7.17所示。AnyNetX_A是一个没有额外限制的原始搜索空间，该空间中第*i*层的搜索参数除了X block中的3个参数(*w_i*, *b_i*和*g_i*)外，还包括该层中块的个数*d_i*，架构的层数设置为4，因此AnyNetX_A空间中待搜索的参数个数，即搜索空间的自由度(degrees of freedom)为4*4=16。AnyNetX_B搜索空间是

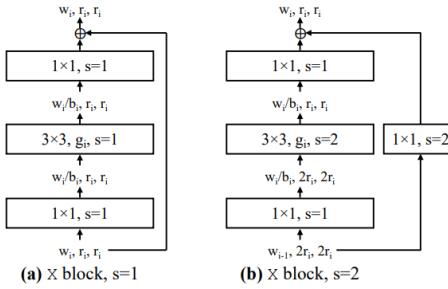


图 7.15 RegNet 搜索空间中块的结构^[257]

AnyNet X_A 的子空间, 其附加了“每一个块具有相同的瓶颈系数 b_i ”的限制, AnyNet X_C 则是进一步附加了“每一个块中的 3x3 分组卷积具有相同的分组数 g_i ”, 通过误差经验分布曲线(图7.16)分析发现, AnyNet X_B 和 AnyNet X_C 在减小了搜索空间大小的同时, 整体性能并未由明显下降。遵循这样的设计流程, AnyNet X_D (附加“块的宽度随着阶段数递增”的限制)和 AnyNet X_E (附加“块的数量随着阶段数递增”的限制)也具有相同的性质。最终的 RegNet 搜索空间是在 AnyNet X_E 的基础上将块的宽度进行线性量化(quantized linear), 进一步减小搜索空间的规模。

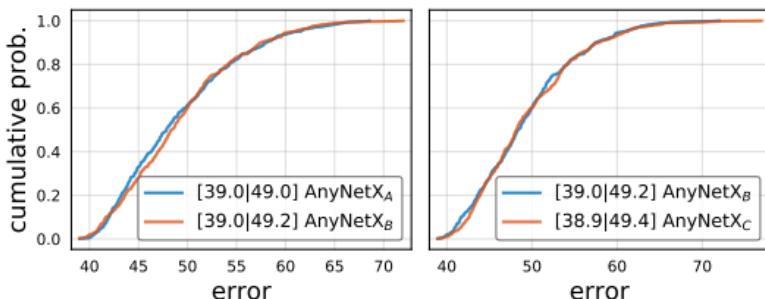


图 7.16 AnyNet_B 和 AnyNet_C 的误差经验分布曲线对比^[257]

RegNet 的设计流程采用逐步缩减的方式，融合了人工判断和自动化设计的思想，通过在较大规模搜索空间中加入合适的参数限制，保证了在不影响整体性能（甚至性能有所提升）的前提下大幅缩减搜索空间的大小，得到一个紧致且质量更高的搜索空间。如图7.17和图7.18所示，RegNet 搜索空间的相比于原始搜索空间 AnyNet X_A ，规模上降低了大约 10 个数量级，但整体性能反而有明显提升。Radosavovic 等人的工作是搜索空间设计领域一次新的尝试，其打破了传统的人工设计搜索空间的设计范式，采

用半自动化的设计流程，得到了一个优秀的搜索空间和部分新颖的设计理念。

	restriction	dim.	combinations	total
AnyNetX _A	none	16	$(16 \cdot 128 \cdot 3 \cdot 6)^4$	$\sim 1.8 \cdot 10^{18}$
AnyNetX _B	$+ b_{i+1} = b_i$	13	$(16 \cdot 128 \cdot 6)^4 \cdot 3$	$\sim 6.8 \cdot 10^{16}$
AnyNetX _C	$+ g_{i+1} = g_i$	10	$(16 \cdot 128)^4 \cdot 3 \cdot 6$	$\sim 3.2 \cdot 10^{14}$
AnyNetX _D	$+ w_{i+1} \geq w_i$	10	$(16 \cdot 128)^4 \cdot 3 \cdot 6 / (4!)^2$	$\sim 1.3 \cdot 10^{13}$
AnyNetX _E	$+ d_{i+1} \geq d_i$	10	$(16 \cdot 128)^4 \cdot 3 \cdot 6 / (4!)^2$	$\sim 5.5 \cdot 10^{11}$
RegNet	quantized linear	6	$\sim 64^4 \cdot 6 \cdot 3$	$\sim 3.0 \cdot 10^8$

图 7.17 RegNet 的设计过程^[257]

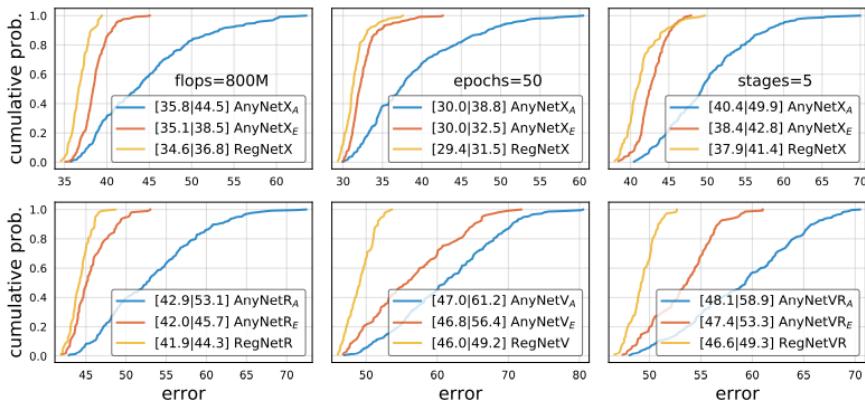


图 7.18 RegNet 的性能分析^[257]

7.3.3 总结

搜索空间作为 NAS 方法中不可或缺的部件，自 Zoph 等人于 2017 年提出 NAS 方法后，便受到了广泛的关注和研究。总结来说，NAS 搜索空间可以分为人工设计搜索空间和自动设计搜索空间两大类，其中，人工设计搜索空间按照其包含的搜索决策，又可以进一步划分为宏观架构搜索空间、微观架构搜索空间、非拓扑架构搜索空间和层次化架构搜索空间。从算法研究的角度看，搜索空间的自动化设计是该研究领域的前沿方向之一，仍具备充足的研究价值和空间。从实际应用的角度看，人工设计的搜索空间得到了更广泛的应用，特别地，在大多数硬件资源受限的任务场景中，为了取得任务性能和硬件开销（例如延时、功耗等）的权衡，非拓扑架构的搜索空间因其硬件友好的性质得到了大部分研究者的青睐。

7.4 搜索策略

搜索策略定义了探索搜索空间的方式，即如何在搜索空间中采样候选架构。其关键在于提高采样效率，即在实际需要评估的架构数目尽可能少的情况下，探索得到性能尽可能高的架构。对于 NAS 算法来说，搜索策略面临着探索与利用（exploration-exploitation）的权衡。一方面，搜索策略需要充分探索搜索空间，在一定程度上鼓励采样预期性能带有不确定性的架构，避免搜索过早地收敛到次优架构；另一方面，搜索策略需要充分利用已经掌握的架构-性能信息，对预期性能较高的架构进行采样，从而避免将过多的评估资源浪费在性能较差的架构上。

TODO: 理解换一个词

搜索策略的功能可以概括为“根据已评估的架构-性能信息，从搜索空间中采样下一个或下一批架构”。具体而言，其功能可拆分为以下两点：1) “理解”，即根据已评估的架构-性能信息理解该搜索空间中具有较高性能的架构模式；2) “运用”，即根据理解的架构模式在搜索空间中采样新架构。根据这两点不同，研究者设计了不同的搜索策略。穷举法是最朴素并且探索搜索空间最充分的搜索策略。然而考虑到实际搜索空间的规模，与之对应的搜索代价往往是不可承受的。当前主流的搜索策略主要包含五类，分别是基于强化学习的搜索（Reinforcement Learning Based Search）^[29, 392, 393]、基于进化算法的搜索（Evolutionary Search）^[71, 259, 260]、随机采样（Random Sampling）^[173]、基于架构性能预测器的搜索（Predictor-based Search）^[215, 238, 313]与可微分架构搜索（Differentiable Search）^[196, 281, 334]。特别地，可微分架构搜索与基于权重共享的评估策略（7.5.2节）密不可分，而其他主流搜索策略则可以与评估策略解耦。

本节将围绕上述提及的主流搜索策略进行展开：7.4.1节将介绍基于强化学习的搜索；7.4.2节将介绍基于进化算法的搜索；7.4.3节将介绍随机采样的搜索策略；7.4.4节将介绍基于架构性能预测器的搜索。特别地，由于可微分架构搜索的独特性，本章将在7.6节单独对其介绍。

7.4.1 基于强化学习的搜索

最早期的搜索策略将 NAS 建模为强化学习问题，应用强化学习算法进行求解，称之为基于强化学习的搜索策略。

基于强化学习的搜索策略实现“运用”功能的方式是“使用生成神经网络架构^[246, 392, 393]或对神经网络架构进行转化^[97]的控制器（Controller）或称智能体（Agent）从搜索空间中采样新架构”。其实现“理解”功能的方式，也是与其他搜索策略最核心的区别在于“应用强化学习算法以生成的神经网络架构的性能分数作为奖励优化控制器”，从

而使控制器能够采样得到性能更强的神经网络架构。

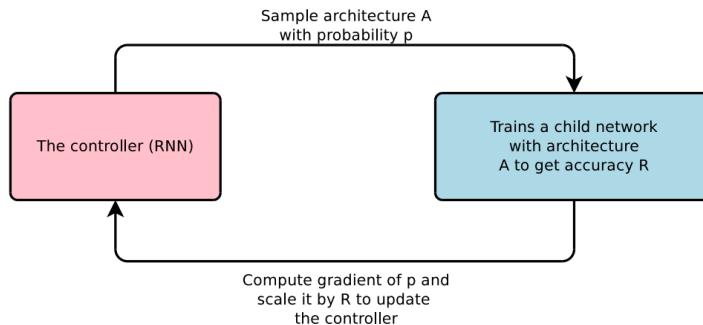


图 7.19 基于强化学习的架构搜索流程^[392]

以 NASNet^[393] 搜索策略为例，其搜索流程如图7.19所示。第一步，应用控制器从搜索空间中以概率 p 采样出候选架构 A ；第二步，将该架构在训练集上进行训练，以训练后在验证集上的性能作为奖励 R ；第三步，根据 p 与 R 计算梯度，更新控制器的权重。具体而言，近端策略优化算法^[272]（Proximal Policy Optimization, PPO）被用于优化控制器。重复上述流程，直到满足迭代结束条件。

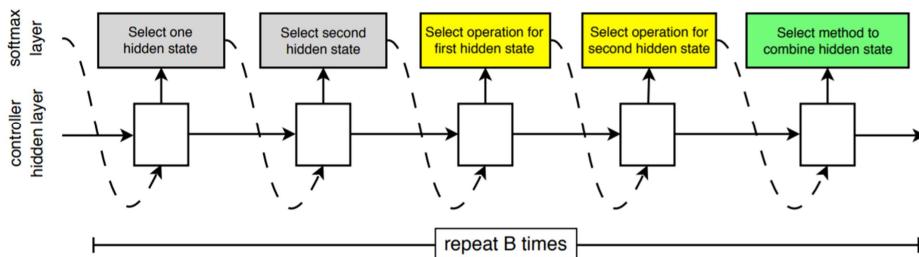


图 7.20 NASNet 控制器架构

基于强化学习的搜索策略使用的控制器通常由多层感知机^[111]或者循环神经网络^[246, 392, 393]构成，并且往往需要针对不同的搜索空间单独进行设计。例如，NASNet 使用基于循环神经网络的控制器生成神经网络卷积单元的架构。如7.3.1节所述，其搜索空间由卷积单元堆叠而成。每个卷积单元包含 B 个模块，每个模块从单元输入或之前模块的输出中选择两个隐状态（hidden state）作为输入，然后选择两种操作分别应用于两个输入，最终选择一种方式结合计算结果作为输出。其控制器的具体架构如图7.20所示。控制器对每个单元架构的预测分为 B 个模块，每个模块有 5 个预测步骤，由 5 个不同的 softmax 分类器完成，对应于一个单元模块架构的离散选择。

TODO: “策略表示”可以换成“控制器结构”。

不同强化学习搜索策略之间的差异性主要在于以下两点: 1) 策略表示。例如, Zoph 等人^[392] 使用循环神经网络顺序采样一个编码架构的字符串; EAS^[29] 提出使用双向 LSTM 将神经网络架构编码为固定长度表示, 以处理变长网络体系结构; 2) 优化方式。例如, Zoph 等人在 2016 年的工作^[392] 中使用强化策略梯度算法^[317] 对控制器进行端到端优化, 而在 NASNet 中使用 PPO 优化控制器; Baker 等人^[13] 使用 Q 学习方法^[310] 学习一个逐层对神经网络层的类型及相应超参数进行选择的策略。

TODO: 这一段换到小节里面, 然后只是推荐进化算法与预测器好了。还是回避有争议的表述吧。

特别要说明的是, 最近的研究^[314] 指出, 基于强化学习的搜索策略的采样效率在公平的比较下被基于进化算法 (7.4.2节) 以及预测器的搜索策略 (7.4.4节) 超越。此外, 该搜索策略使用的控制器的架构以及训练超参数均需要针对不同的搜索空间手工调整, 并且研究者难以在搜索过程中进行诊断或人为干预。因此, 该搜索策略在最近研究中的实际应用不多。

7.4.2 基于进化算法的搜索

进化算法在 NAS 中的应用可以追溯到 1989 年, Miller 等人^[224] 使用遗传算法设计网络架构并使用反向传播算法优化权重。此后, 许多研究^[7, 282, 283] 提出同时使用进化算法优化网络架构与权重。但是随着基于 SGD 的权重优化算法极大地提升了神经网络的性能, 近年来的研究^[71, 259, 260] 重新转向使用梯度方法优化权重并使用进化算法优化神经网络架构。

基于进化算法的搜索策略实现“理解”功能的方式在于维持一个种群 (population), 即一组候选架构。根据不断采样的新架构的信息, 该搜索策略对种群进行更新, 逐步淘汰掉性能较差的架构, 保留性能较高的架构。其实现“运用”功能的方式是“从种群中以某种规则采样出部分架构作为亲本 (parent), 并应用突变 (mutation)、交叉 (crossover) 等操作生成一个或多个新的后代架构 (child)”。

TODO: 加粗下划线少一些

基于进化算法的搜索流程通常可以划分为两个阶段。第一阶段, 基于进化算法的搜索策略会从搜索空间中随机采样一定数目的神经网络架构, 应用评估策略对这些架构的性能进行评估, 然后将其作为初始种群。第二阶段, 基于进化算法的搜索策略会对种群进行迭代。每一轮迭代可分解为以下三个步骤: 1) 从种群中采样亲本: 以某种方式从种群中采样一个或多个性能较好的架构作为亲本; 2) 从亲本产生后代: 应用突变、交叉等操作从亲本生成一个或多个后代架构, 并应用评估策略对这些后代的性

算法 7.1 Aging Evolution 算法流程

```

1: population  $\leftarrow$  empty queue
2: history  $\leftarrow \emptyset$ 
3: while  $|population| < P$  do
4:   model.arch  $\leftarrow$  RANDOM_ARCHITECTURE()
5:   model.accuracy  $\leftarrow$  TRAIN_AND_EVAL(model.arch)
6:   add model to right of population
7:   add model to history
8: end while
9: while  $|history| < C$  do
10:   sample  $\leftarrow \emptyset$ 
11:   while  $|sample| < S$  do
12:     candidate  $\leftarrow$  random element from population
13:     add candidate to sample
14:   end while
15:   parent  $\leftarrow$  highest-accuracy model in sample
16:   child.arch  $\leftarrow$  MUTATE(parent.arch)
17:   child.accuracy  $\leftarrow$  TRAIN_AND_EVAL(child.arch)
18:   add child to right of population
19:   add child to history
20:   remove dead from left of population
21:   discard dead
22: end while
23: Return highest-accuracy model in history

```

能进行评估；3) 更新种群: 以某种方式对种群进行更新。最终，搜索策略会返回历史或种群中性能最佳的一个或多个架构作为搜索结果。

以 Real 等人^[259] 提出的带有老化机制的进化算法 (Aging Evolution) 为例，其整体流程如算法7.1所示。在搜索的第一阶段 (行 1-8)：该搜索策略从搜索空间中随机采样 P 个候选架构，训练每个架构并测试其性能 (行 5)，然后将所得架构加入种群与历史 (*history*) (行 6-7)；在搜索的第二阶段 (行 9-22)：该搜索策略对种群进行迭代。每轮迭代中，该搜索策略先从种群中均匀随机抽样 S 个架构，选择其中具有最佳性能的架构作为亲本 (*parent*) (行 10-15)。接着，该策略通过突变从该亲本生成子代架构 (*child*, 行 16)，并对其进行性能评估 (行 17)，然后将其添加到种群与历史之中 (行 18-19)。当历史中的架构数目达到预先设定的数目 C 时，结束搜索，返回历史中性能最佳的架构作为搜索结果。特别地，Aging Evolution 引入了独特的老化机制，以鼓励对搜索空间更好的探索。具体而言，在搜索的第二阶段，种群保持架构数目不变。该策略每次将新架构添加进种群后，种群中最早采样得到的架构 (*dead*) 都会被移出种群 (行 20-21)。

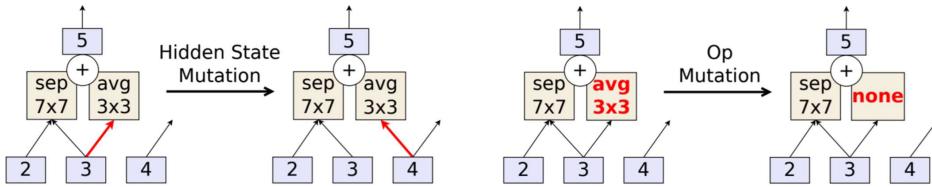


图 7.21 神经网络架构突变示例。左图中，“avg 3x3”操作的输入节点从“3”变为“4”。右图中，“avg 3x3”操作被替换为“none”。

突变是一种局部操作。在 NAS 中，常见的突变操作包括添加或删除一个层、改变一个层的超参数、添加跳跃连接等。图7.21展示了两个突变的例子。其中“2”、“3”、“4”为该层的输入节点，“5”为输出节点， 7×7 离散卷积（“sep 7×7 ”）与 3×3 平均池化（“avg 3×3 ”）为该层的两个候选操作。两操作的输出结果加和并传递至节点“5”。左图中的突变将池化操作的输入节点从“3”变为“4”，右图中的突变将平均池化替换为无（“none”，即没有输出）。由此可见，进化算法中的突变操作，在设计上类似于生物学中的突变。这正是其名字的由来。

为了更好地探索搜索空间，基于进化算法的搜索策略主要有以下两点设计：1) 从种群中选择亲本的方式。大多数进化算法在产生下一代时都避免仅选择种群中的最佳个体作为亲本，而是在倾向于选择种群中性能更好的个体的同时，以某种方式引入一定的随机性。例如，Aging Evolution 将从种群中随机采样的 S 个架构中的最佳架构作为亲本，而不是将整个种群中的最佳架构作为亲本；2) 从亲本生成子代的方式。出于同样的原因，突变、交叉等操作将多样性引入种群，通过防止种群中的个体彼此过于相似，避免陷入局部最优。

TODO: 加入 Aging Evo 的机制也是一种探索搜索空间的方式。

如前文所述，基于进化算法的搜索策略对种群的迭代可分解为三个步骤。不同搜索策略的差异主要就在于实现这三个步骤的方式：1) 从种群中采样亲本的方式。例如，Real 等人^[260] 提出竞赛选择的方式从种群中采样，而 Elsken 等人^[71] 则使用逆密度从多目标帕累托前沿对亲本进行采样；2) 从亲本产生后代的方式。从亲本产生后代这一过程包含架构与权重两个维度。突变与交叉是产生后代架构的主要方式，其中突变已在前文加以阐述。交叉指的是交换两个或多个亲本的部分架构，不同方法的具体做法往往略有不同。对于如何生成后代架构的权重，多数方法通过随机初始化网络来产生后代，但是这带来了极大的评估开销。Elsken 等人^[71] 提出采用拉马克继承，即通过网络态射（morphisms）将知识从亲本网络传递给后代网络，而 Real 等人^[260] 则让后代架构继承其亲本不受突变影响的所有参数；3) 更新种群的方式。例如，Real 等人在 2017 年的工作^[260] 中从种群中去除性能最差的架构，而在 2019 年的工作 Aging Evolution 中

发现去除最年长的个体是有益的，Liu 等人^[194]则不对种群中的架构进行去除。

7.4.3 随机搜索

随机搜索策略是最朴素的搜索策略之一，广泛作为相关研究的基准方法。

随机搜索策略并没有针对“理解”与“运用”的功能设计。顾名思义，该搜索策略不会根据已经评估的架构-性能信息理解任何架构模式，在搜索过程中随机采样架构。因此，随机搜索策略往往不可避免地采样大量性能较差的架构，采样效率较低。但是从另一个角度来看，这也避免了随机搜索陷入局部最优。

算法 7.2 随机搜索算法流程

```

1: history  $\leftarrow \emptyset$ 
2: while  $|history| < C$  do
3:   model.arch  $\leftarrow$  RANDOM_ARCHITECTURE()
4:   model.accuracy  $\leftarrow$  TRAIN_AND_EVAL(model.arch)
5:   add model to history
6: end while
7: Return highest-accuracy model in history
```

典型的随机搜索流程如算法7.2所示。算法从搜索空间中随机采样并评估 C 个神经网络架构（行 2-6），最终返回其中评估分数最高的架构作为搜索结果。

由于随机搜索的采样效率较低，Li 等人^[173]在搜索过程中应用基于权重共享的单次评估策略以降低整体搜索开销。基于权重共享的单次评估策略构建一个过参数化的超网络 (SuperNet)，使得搜索空间中所有的架构都是该超网络的子网络。该评估策略将超网络在训练数据集上训练至收敛后，就可以使用每个候选架构在超网络中对应的子网络权重进行性能评估，从而避免从头训练候选架构。该评估策略将在7.5.2节详述。此外，Li 等人还结合早停机制^[172]对随机搜索策略进行了改进。对这部分内容感兴趣的读者可以参考原文。

Li 提出的搜索方法在 CIFAR-10 与 PTB 数据集上可以达到甚至超过 ENAS^[246]（一种基于强化学习的搜索策略）的性能。而在 NAS-Bench-201^[67] 上，随机搜索得到的架构也被发现显著优于可微分搜索 DARTS^[196] 得到的架构。

7.4.4 基于架构性能预测器的搜索

基于架构性能预测器的搜索策略实现“理解”功能的方式在于学习一个近似的**架构性能预测器**。如图7.22所示，架构性能预测器一般以架构表示（如邻接矩阵^[354, 392]、路径编码^[311, 313]）为输入，以预测性能分数为输出，由架构编码器与预测头两部分构成。

其中，架构编码器将输入的架构表示编码为连续空间的隐表征，预测头则将隐表征映射到预测分数。

基于架构性能预测器的搜索策略实现“运用”功能的方式是在搜索过程中利用该架构性能预测器给出的架构分数进行快速评估，并采样预测分数较高的架构进行实际评估，从而降低更慢、更准确的评估策略需要评估的架构数量。由于架构性能预测器一般可以在毫秒量级的时间内给出一个架构的近似性能评估结果，因此整体搜索十分高效。

TODO: 就保留

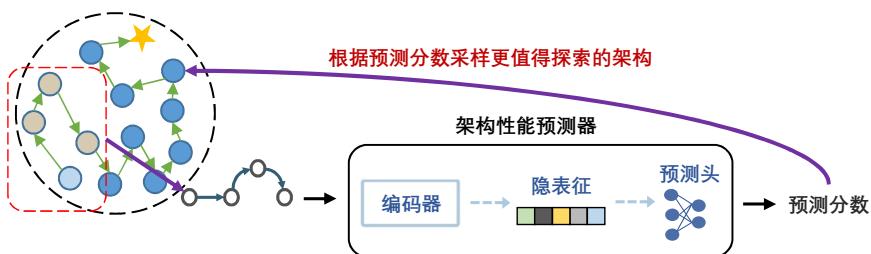


图 7.22 基于架构性能预测器的搜索流程示意图

基于预测器的 NAS 的典型流程如算法 7.3 与图 7.23 所示。每轮迭代中，首先基于预测器的预测结果从搜索空间中采样一定量 ($N^{(k)}$) 的架构 (行 6)，然后应用评估策略对这些新采样出的架构进行评估 (行 7)。接着，这些新采样出的架构与其相应的评估结果被加入训练架构集以微调预测器 (行 8)。迭代结束后，返回所有评估过的架构中性能最佳的架构或者当前预测器认为搜索空间中的最佳架构作为最终架构。

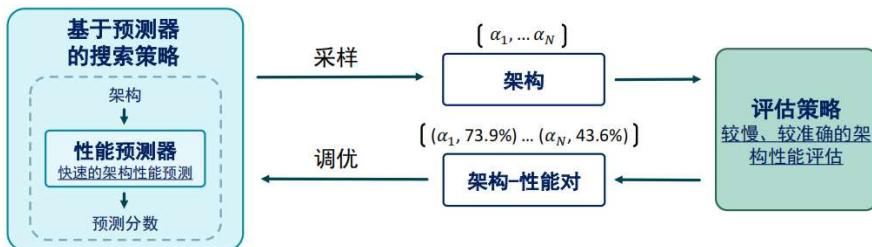


图 7.23 基于架构性能预测器的搜索流程示意图

基于预测器的预测结果从搜索空间中进行采样的方法大致分为以下两类：1) 基于离散搜索的采样策略，如 GATES^[238]、BANANAS^[313]。这种策略以预测器的预测分

算法 7.3 基于预测器的架构搜索流程

```

1:  $\mathcal{A}$ : 架构搜索空间
2:  $P : \mathcal{A} \rightarrow \mathbb{R}$ : 架构性能预测器, 输入架构, 输出架构的预测性能
3:  $N^{(k)}$ : 第  $k$  个循环需要采样、评估的架构个数

4:  $k = 1$ 
5: while  $k \leq \text{MAX\_ITER}$  do
6:   从搜索空间  $\mathcal{A}$  里, 利用预测器  $P$  的预测, 采样  $N^{(k)}$  个架构  $S^{(k)} = \{a_j^{(k)}\}_{j=1,\dots,N^{(k)}}$ 
7:   将新采样的架构  $S^{(k)}$  进行评估, 得到评估结果 (架构-评估性能对)  $\tilde{S}^{(k)} = \{(a_j^{(k)}, y_j^{(k)})\}_{j=1,\dots,N^{(k)}}$  ( $y$  是评估性能)
8:   用已有的所有架构-评估性能对数据  $\tilde{S} = \bigcup_{i=1}^k \tilde{S}^{(i)}$  优化预测器  $P$ 
9: end while
10: return 所有评估过的架构里有着最高评估性能  $y_{j^*}$  对应的  $a_{j^*} \in \bigcup_{i=1}^k S^{(i)}$ , 或者输出预测器认为最好的架构  $a^* = \operatorname{argmax}_{a \in \mathcal{A}} P(a)$  作为搜索结果

```

数为奖励, 在“内部”进行离散搜索以找到预测分数较高的架构集合。常用的内部离散搜索方法包括穷举、随机搜索、进化算法等; 2) 基于梯度的采样策略^[175, 215]。这种策略基于预测器的预测分数对架构表征求梯度并优化。例如, NAO^[215] 更新架构隐表征, 然后使用架构解码器将更新后的架构隐表征映射到架构。而 Li 等人^[175] 则直接对架构表征进行更新。

基于架构性能预测器的搜索策略面临的核心问题在于架构性能预测器需要一定量的“架构-性能”数据进行训练后才能指导采样。在搜索初期, 架构性能预测器对于数据的需求更加明显。然而, “架构-性能”数据的获取代价通常是很大的。因此, 在实践中能够用于训练初始预测器的“架构-性能”数据往往极为有限。这种情况下, 预测器难以学习到泛化性强的预测能力, 不能给出准确的性能估计值或正确的架构性能排序, 从而损害了采样效率。

当前研究主要从以下三个方面对基于预测器的搜索策略进行改进, 以提高有限训练数据下预测器的泛化能力, 从而提升采样效率。

一、设计更好的架构编码器

架构性能预测器的预测能力与架构编码方案息息相关。一方面, 合理的架构编码方案可以从架构表示中充分提取有效信息。另一方面, 合理的架构编码方案可以引入对性能预测有帮助的先验知识, 进一步提升有限训练数据量下的预测能力。因此, 研究者们尝试设计更好的架构编码器, 以降低预测器对于训练数据量的需求。

如图7.24所示, 现有的神经网络架构编码方案主要包括以下两种编码方式。

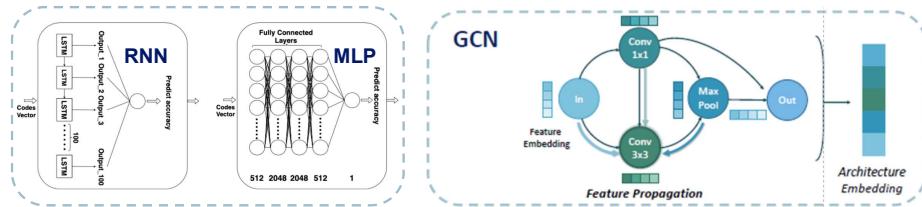


图 7.24 神经网络架构编码方案示意图。左：非图编码；右：基于图的编码。

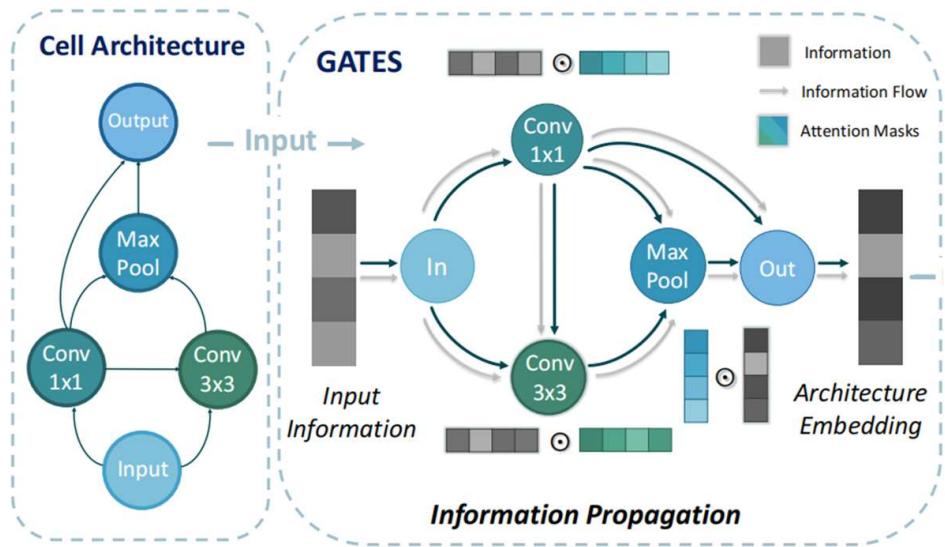


图 7.25 神经网络架构编码器 GATES 图示

非基于图的编码方式 首先使用特定方法将神经网络架构转换为序列^[193, 214, 215, 315] 或者图像^[333]，然后将其输入到 XGBoost、MLP、LSTM 或者 CNN 中进行编码。然而，非基于图的编码方式不能有效地保持架构的拓扑信息，只能依靠学习过程隐式地对拓扑信息进行建模，且对同构架构的处理明显不佳。

基于图的编码方式^[238, 373] 则应用图卷积网络编码神经网络架构。但现有的基于图的架构编码器也存在不合理之处^[238]。相关改进工作例如，GATES^[238] 和 D-VAE^[374] 提出基于图神经网络模拟架构实际处理数据时的信息流对架构进行编码。特别地，最近也有研究^[210, 338] 尝试用 transformer 对架构进行编码。

本小节以 Ning 等人^[238] 提出的架构编码器 GATES 为例，讲述编码器对架构的具体编码过程。GATES 是一种基于图的架构编码器，其核心思想是模拟架构的信息流动过程对架构进行编码。如图 7.25 所示，该编码器的输入是一个用于表示架构的有向无

环图。GATES 模拟虚拟信息 (input information) 通过架构处理的过程。虚拟信息传播至每个操作 (如 Conv 3×3) 时, GATES 都会在该虚拟信息上乘以该操作对应的可学习掩膜 (mask), 以此模拟中间特征通过该操作节点的计算。最终, 经过所有操作 “处理” 过后的虚拟信息将作为该架构的编码结果。

由于 GATES 模拟信息处理过程进行架构编码, 同构架构自然可以被映射到一样的编码。相比于使用普通的 GCN, GATES 的设计思想使其更适合编码数据流图类型数据, 操作被编码成了对节点传递信息的变换, 而不是节点之间传递的信息本身。然而, 神经网络架构不仅描述了正向传播中数据的流动与处理, 还决定了模型训练时的动态学习过程。GATES 仅对前者进行了建模, 没有考虑训练过程对架构性能的影响。因此, 后续的改进方法 TA-GATES^[240] 在编码架构时进一步模拟了神经网络的迭代训练过程及参数随机初始化过程。有兴趣的读者可以阅读其原文。

二、改进架构性能预测器的训练过程

除设计更好的架构编码器, 一些研究着眼于改进架构性能预测器的训练过程以提升预测器在有限数据下的预测能力。

设计与优化目标更加贴合的损失函数。早期研究应用回归损失函数 (如均方误差 MSE) 训练架构预测器, 以最小化预测结果与真实评估结果间的差异。但是架构性能预测器的关键往往不在于预测结果与真实结果的数值差异, 而在于预测结果与真实结果的排序相关性。因此, 后续研究设计了排序损失函数^[238, 333] 来训练架构性能预测器, 从而使用同等数量的训练数据时架构性能预测器可以得到更好的排序相关性。

应用半监督、无监督等先进的训练方法。传统方法以监督学习的方式训练架构性能预测器, 在有限的数据量下泛化性不佳。借鉴深度学习其他领域的研究进展, 研究者尝试应用更先进的训练方法改进预测器的训练过程。例如, SemiNAS^[214] 等方法提出以半监督的方式训练预测器; arch2vec^[339] 提出应用无监督方法对预测器进行预训练以提升编码器的架构表征能力。

TODO: 上面两点合成一点吧

引入额外信息辅助架构性能预测器的学习。上述方法均基于有限的 “架构-性能” 数据充分挖掘预测器的预测能力。近来, 部分研究尝试引入 “架构-性能” 数据外的其他信息以进一步提高预测器的预测能力。这些信息包括单次评估指标 (7.5.2节)、零次评估指标 (7.5.4节) 以及硬件相关的指标 (如参数量、FLOPs) 等。一方面, 这些额外信息中包含的架构排序信息可以帮助预测器获得更强的排序性能。另一方面, 学习拟合额外信息可以鼓励预测器提取更好的架构表征。DELE^[380] 将这些易于获得的额外信息指标称为 “低保真度指标”, 并提出了一套利用多种低保真度指标辅助预测器

表 7.1 不同搜索策略实现“理解”与“运用”功能的方式

搜索策略	“理解”	“运用”
基于强化学习的搜索	应用强化学习算法以生成的神经网络架构的性能分数作为奖励优化控制器	使用生成神经网络架构或对神经网络架构进行转化的控制器从搜索空间中采样
基于进化算法的搜索	维持并更新一个种群	从种群中采样部分架构作为亲本，并应用突变、交叉等操作生成后代架构
随机搜索	-	-
基于架构性能预测器的搜索	学习一个近似的架构性能预测器	用架构性能预测器给出的架构分数进行快速评估，并采样预测分数较高的架构

训练的框架。GATES⁺⁺^[239]则将零次评估指标编码进架构表示，并在训练过程中以多任务学习的形式同时拟合真实性能以及零次评估指标。

表 7.2 部分神经架构搜索研究工作总结

研究工作	搜索策略	评估策略	研究工作	搜索策略	评估策略
ENAS ^[246]	强化学习	权重共享	DARTS ^[196]	可微分搜索	权重共享
NASNet ^[393]	强化学习	独立从头训练	SGAS ^[168]	可微分搜索	权重共享
SPOS ^[98]	进化算法	权重共享	DRNAS ^[44]	可微分搜索	权重共享
AmoebaNet ^[259]	进化算法	独立从头训练	SDARTS ^[43]	可微分搜索	权重共享
Once-for-All ^[30]	预测器	权重共享	DARTS- ^[55]	可微分搜索	权重共享
NAO ^[215]	预测器	独立从头训练	PC-DARTS ^[334]	可微分搜索	权重共享
SemiNAS ^[214]	预测器	独立从头训练	RDARTS ^[361]	可微分搜索	权重共享
BANANAS ^[313]	预测器	独立从头训练	DARTS+PT ^[307]	可微分搜索	权重共享
NEPNAS ^[311]	预测器	独立从头训练	Single Path NAS ^[281]	可微分搜索	权重共享
GATES ^[238]	预测器	独立从头训练	Fair DARTS ^[56]	可微分搜索	权重共享

三、不确定度建模

有研究提出在搜索过程中显式对不确定性进行建模，鼓励采样不确定性较大的架构区域，从而更充分地探索搜索空间。例如，BANANAS^[313]用同样的数据训练多个预测器，并将多个预测器对同一个架构的预测结果的标准差做为不确定性的度量；GP-NAS^[181]采用高斯过程建模架构与其性能的关系。

7.4.5 小结

搜索策略是神经架构搜索系统的核心组件之一，面临着探索与利用的权衡。一方面，搜索策略需要充分探索搜索空间，采样预期性能带有不确定性的架构；另一方面，搜索策略需要利用已有的信息对预期性能较高的架构进行采样。它的功能可以总结为“根据已评估的架构-性能信息，从搜索空间中采样下一个或下一批架构”。根据实现“理解”与“运用”功能的方式的差异，研究者设计了不同的搜索策略。表7.1对常见的搜索策略进行了总结。在本节最后，表7.2给出了神经架构搜索领域部分知名工作对应的搜索策略以及评估策略。

7.5 评估策略

在神经网络架构搜索的过程中，**评估策略的作用是评估架构的性能表现，为搜索策略提供指导性信息**。例如，在7.4.1节介绍的基于强化学习的搜索策略中，评估策略给出的架构性能将直接作为控制器的奖励 R ；在7.4.2节介绍的基于进化算法的搜索策略中，架构的性能将作为从种群中挑选优秀个体、产生后代的主要指标。在搜索过程中，若评估策略提供不准确的性能结果，控制器将会很大概率会偏离最优的搜路径。换言之，评估策略对架构性能评估的准确性，将直接影响最终搜索结果的性能好坏。

独立训练 (stand-alone training) 策略是一种最为直接的评估策略，它通过完整地对模型架构进行训练和测试得到架构的性能。该评估策略具有较高的准确性，常被用来作为基准结果，但其效率非常底下，对 GPU 资源和时间资源的需求量往往令人难以接受。对此，有研究工作设计规模较小的代理任务来减小巨大的资源开销，例如减小待训练模型大小^[259]、缩小训练数据集^[319]的规模、减少训练轮次^[393]或是通过权重继承的方式加速模型拟合^[29]。然而，这些研究结果表明，代理任务对独立训练策略的开销优化有限，整体的评估开销依然很大，例如，NASNet^[393]将训练轮次降低为 20 次，但仍在 CIFAR-10 数据集上花费了 2000 个 GPU 小时完成搜索任务。

单次评估 (one-shot estimation) 策略是一种较为高效的评估策略，相比于独立训练策略需要上千 GPU 天来完成，单次评估策略将总训练时长降低到了小时量级。单次评估策略主要有两种实现方式：一种是使用权重共享的超网络 (SuperNet) 均摊模型架构的训练开销，该方式易于实现，且性能较为优秀，是目前最常用的方式；另一种是使用超网络 (HyperNet) 为待评估架构直接生成权重，一定程度上可以看作第一种实现方式的泛化方案，但准确地生成模型权重是一个较为困难的任务，因此该方案的应用范围较小。单次评估策略尽管相比于独立训练有显著的效率提升，但对架构性能的评估较为不准确，一些研究^[16, 213, 236, 356, 361, 390]从不同角度分析了评估不准确的原因，并提

出了优化改进的方案。总体来说，单次评估策略在精度和效率之间达到了很好的平衡，因此在学术界和工业界均得到了广泛使用。

零次评估 (zero-shot estimation) 策略是目前速度最快的评估策略，该策略完全或几乎不需要架构的训练，只需要对一个随机初始化的架构进行一次前向计算即可获取相应的评估指标，因此其对架构的评估开销仅在毫秒量级。然而，与单次评估策略相比，零次评估策略更大程度地牺牲了评估的准确性，因此，该策略目前仍处于研究阶段，还难以被应用在实际场景中。

本节将对上述的三种评估策略展开介绍：7.5.1节将介绍独立训练策略及其优化技巧；7.5.2节将着重介绍基于权重共享的单次评估策略，具体来说，我们将分别从其基本流程、存在的问题和改进方案展开讨论；7.5.3将简要地介绍基于权重生成的单次评估策略；7.5.4节将介绍若干种零次评估策略的评估指标，并简单展示其在性能方面的缺陷。

7.5.1 独立训练策略

独立训练策略(如图7.26所示)的流程是，在训练数据集上从头到尾训练模型架构至收敛，然后在验证数据集上对模型进行测试。最早期的工作^[392] 使用了该策略，为了评估一个架构的性能，研究人员将架构在CIFAR-10数据集上训练了50轮，并且整个搜索过程中评估了上万个架构，最终消耗48000个GPU小时才完成了搜索任务。

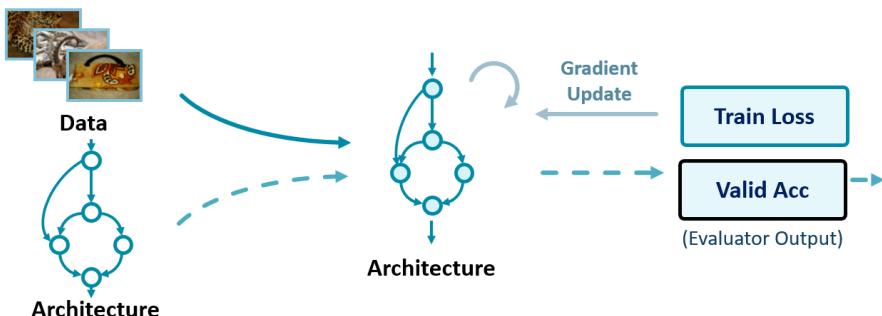
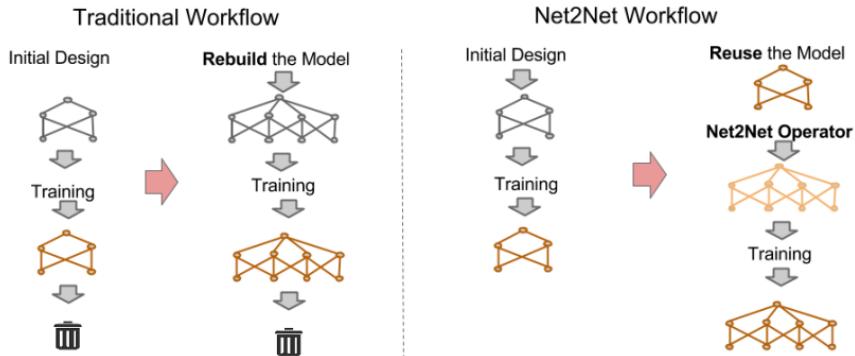
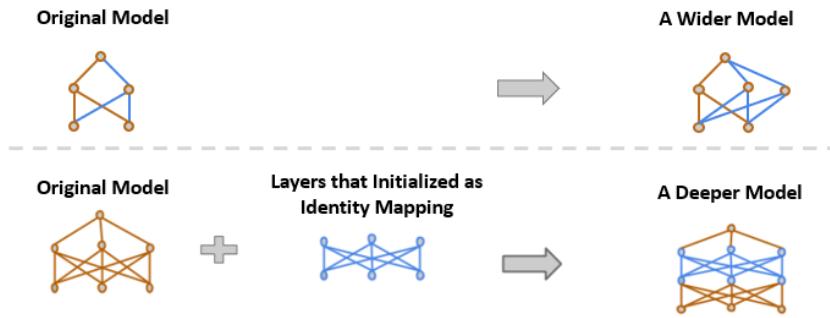


图 7.26 独立训练策略的流程

独立训练策略的时间开销十分庞大，最直接的原因是在任务数据集上需要将架构从头训练至收敛。针对这一问题，一些研究通过提出设计规模较小的代理任务，用代理任务的性能间接评估原始任务的性能。具体来说，代理任务可以分为三种类型：(1) **代理模型 (proxy model)** 是指对原始模型架构按照某种规则进行缩小得到的模型，将对代理模型的评估性能用于评估原始模型，由于缩小后的模型更容易训练收敛，因此

图 7.27 独立训练策略和基于网络变换的评估策略对比^[41]图 7.28 两种网络变换的方式: 上半部分示意对网络某一层的加入新的神经元使其拓宽, 下半部分示意在网络中加入新的层使其变深^[41]

评估的开销得到缩减。Real 等人^[259]提出了代理模型的技巧, 特别地, 他们减少了待评估模型包含的单元数量和网络每一层的输出通道数量, 值得注意的是, 代理模型往往保留原始模型的拓扑结构, 而会在非拓扑的维度上进行缩减。(2) **代理数据集 (proxy dataset)**是指对原始的任务数据集进行缩小得到的数据集, 例如, Wu 等人^[319]随机选取了 ImageNet 图象分类数据集中的 100 个类别, 以此构建了一个规模较小的代理数据集进行模型架构的训练和评估。(3) **代理训练 (proxy training)**是指通过改变原始训练的参数得到新的训练流程, 例如, Zoph 等人^[393]采用了早停 (early stop) 策略, 对于每个架构只训练 20 个轮次, 较大程度地缩减了训练时长。

除了上述设计代理任务加速评估的方法外, Cai 等人^[29]设计了一种基于网络变换的进化算法 EAS。具体来说, EAS 的核心思想是, 在进化算法的变异过程中, 对父代

架构通过特定的网络变换方式产生子代架构，并让子代架构模型继承父代架构模型的权重，从而加速子代架构模型的训练过程。图7.27对比了传统的独立训练策略和EAS的评估策略，独立训练策略会单独地从头训练每一个待评估的架构，且每次训练评估结束后，训练好的模型权重会被抛弃，而在EAS算法中，子代架构模型会继承父代架构模型的训练权重，这使得子代架构模型的训练只需要快速的权重微调即可，很大程度上节省了训练评估的开销。为了使得子代架构模型能最大程度地复用父代架构模型的权重，EAS借鉴Chen等人的工作^[44]，采用了两种功能保留(function-preserving)变换方式。如图7.28所示，其中一种变换方式是对网络中的某一层进行拓宽(Net2Wider)，即增加卷积层的滤波器(filter)数量或全连接层地单元(unit)数量，新增加的滤波器或单元从原有的滤波器或单元之中随机选择一个来复制权重。另一种变换方式是对网络进行加深(Net2Deeper)，即在网络的某个位置插入一个卷积层或全连接层，对于新的卷积层，将其滤波器初始化为恒等滤波器，而对于新的全连接层，将其权重矩阵初始化为恒等矩阵。通过上述变换得到的新网络能很大程度地保留了旧网络在目标任务上的性能，因此新网络只需要简单的权重微调即可收敛，很大程度地降低了训练评估开销。

7.5.2 基于权重共享的单次评估策略

基于权重共享的单次评估策略(以下简称权重共享策略)最早由Pham等人^[246]提出(如图7.29所示)，其基本流程是，构建一个过参数化的超网络(SuperNet)，使得搜索空间中所有的架构都是该超网络的子网络，将超网络在训练数据集上训练至收敛，评估一个架构的性能时，直接使用该架构对应的子网络权重进行性能测试。

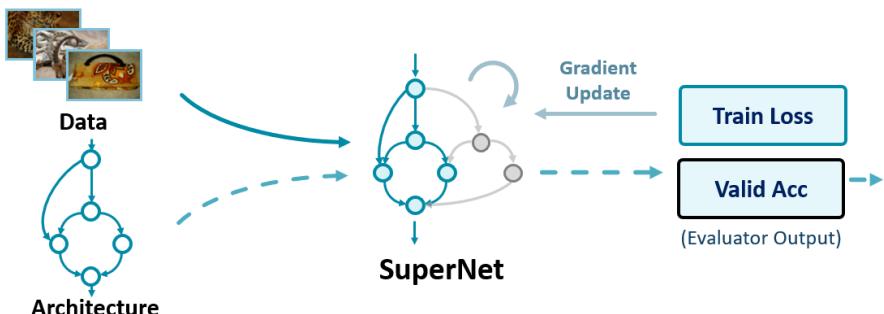


图 7.29 基于权重共享的单次评估策略

超网络对应一个搜索空间中“最大”的网络结构。图7.30展示了在一个包含6个结点的微观架构搜索空间(定义请参考7.3.1节)中构建和使用超网络的例子。其中，超网

络包含了所有可能的连接方式，即任一搜索空间中的网络均为该超网络的子网络，权重共享机制便是通过超网络对所有可能的子网络进行权重共享。例如，图中所示的两个子网络 SubNet-1 和 SubNet-2 均从该搜索空间中采样得到，它们共享了超网络的权重 $\mathcal{W}_{21}, \mathcal{W}_{42}, \mathcal{W}_{63}$ 。

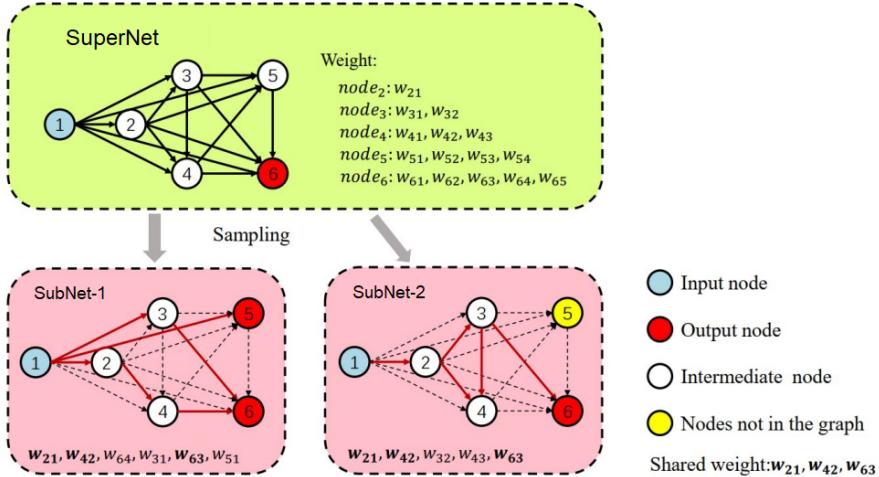


图 7.30 超网络的共享权重机制^[263]

超网络的训练效果直接影响权重共享策略的准确性。具体来说，用 A 表示搜索空间， a 表示其中的架构， W 表示超网络的权重， $W(a)$ 表示架构 a 对应于超网络上的子网络权重， $\mathcal{N}(a, W(a))$ 表示一个具有权重的神经网络模型，则超网络的训练可以用式 7.5 表示，其中， \mathcal{L}_{train} 为模型在训练数据集上的损失函数值。由于计算所有模型的损失期望难以实现，一种主流的训练方式是，随机采样一个架构，将该架构对应的子网络进行一步的权重更新，重复该步骤直至超网络收敛。该方式形式简介，易于实现，且被研究^[98] 证明具有较好的收敛性和较稳定的训练效果。

$$W = \underset{W}{\operatorname{argmin}} \underset{a \in A}{E} [\mathcal{L}_{train}(\mathcal{N}(a, W(a)))] \quad (7.5)$$

相比于在独立训练策略需要对大量架构进行耗时的训练，权重共享策略仅需要训练一个超网络，其总体的时间开销得到了大规模的缩减。AmoebaNet^[259] 使用独立训练策略搭配进化算法，整个搜索过程耗时 3150 个 GPU 天；而 ENAS^[246] 使用权重共享策略搭配进化算法，仅使用 1 个 GPU 天便搜索得到高性能的网络模型；随后，CARS^[351] 对进化算法进行改进，搭配高效的权重共享策略，仅使用 0.4 个 GPU 天探索到高性能的模型架构。

尽管权重共享策略对 NAS 过程的加速效果显著，但其本身存在着准确性差的问题，准确地说，单次评估策略给出的架构性能与架构真实的性能之间排序不一致，这一问题会导致在搜索过程中可能会错误地排除掉优秀的架构，或是选择了性能次优的架构。相关研究^[16, 213, 236, 340]通过充分的实验证明，权重共享策略的性能评估结果与真实性能间的排序相关性（以下简称评估相关性）较差，而相关性差的评估结果会使得搜索路径发生偏差，导致最终搜索到的架构性能次优。在接下来的部分，笔者将以两篇代表性的工作为例，展示权重共享策略存在的具体问题，分析产生问题的原因，以及介绍可行的改进方向和研究工作。

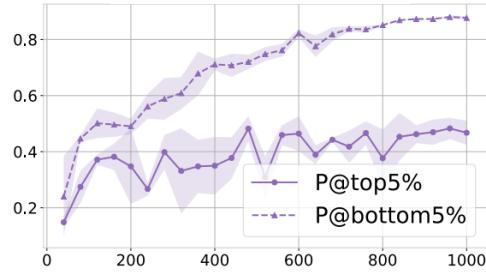
问题与原因分析

本小节以 Ning 等人^[236]的研究工作 EPEE 为例，展示和分析权重共享策略存在的问题。该工作在 5 个 NAS 基准集上验证了权重共享策略的评估质量，包括 NAS-Bench-101^[354]（以下简称 NB101）、NAS-Bench201^[67]（以下简称 NB201）、NAS-Bench-301^[274]（以下简称 NB301）、NDS-ResNet^[256] 和 NDS-ResNeXt^[256]，使用 7 种评估指标对其进行分析，是目前分析维度最全面的研究工作之一。考虑到本书的篇幅原因，本书仅选取 3 种具有代表性的评估指标进行介绍。为了方便读者理解，不妨设共有 M 个架构 $\{a_i\}_{i=1}^M$ ， $\{y_i\}_{i=1}^M$ 和 $\{s_i\}_{i=1}^M$ 分别表示架构的真实性能和评估性能， $\{r_i\}_{i=1}^M$ 和 $\{n_i\}_{i=1}^M$ 分别表示架构按照真实性能的排名和按照评估性能的排名，则上述三种指标分别为：

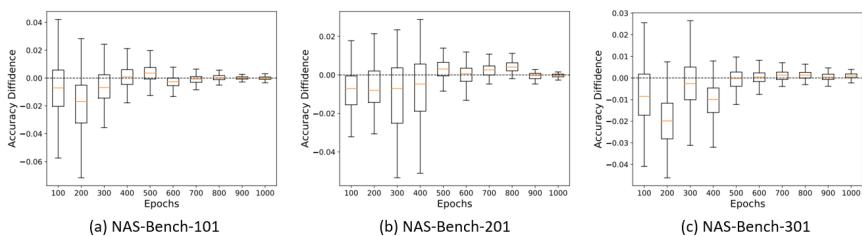
1. 肯德尔排序相关性系数 (Kendall's Tau, 以下简称 KD 系数)：指同序对和逆序对的对数之差与总对数的比值，即 $\sum_{i < j} sgn(y_i - y_j) sgn(s_i - s_j) / \binom{M}{2}$ 。
2. 头部架构的评估准确率 (Precision@topK, 以下简称 P@topK)：指真实性能排名前 K 名的架构占评估性能排名前 K 名架构的比例，即 $\frac{\#\{i | r_i < KM \wedge n_i < KM\}}{KM}$ 。
3. 尾部架构的评估准确率 (Precision@bottomK, 以下简称 P@bottomK)：指真实性能排名后 K 名的架构占评估性能排名后 K 名架构的比例，即 $\frac{\#\{i | r_i > (1-K)M \wedge n_i > (1-K)M\}}{KM}$ 。

通过对大量实验结果的观察，作者证实了权重评估策略存在严重的评估相关性差的问题，同时还总结了权重共享策略存在的两个关键现象：(1) 对超网络做更充分的训练能有效提升其评估相关性，但在大规模的搜索空间中相关性系数依然较低；(2) 权重共享策略对尾部架构（性能较差）的评估准确率高于对头部架构（性能较好）的评估准确率。例如，在 NB201 上 P@topK 显著低于 P@bottomK（如图 7.31 所示）。

针对上述问题，作者做了进一步的实验分析，从评估偏差 (bias) 和评估波动 (variance) 两个方面，总结了导致评估相关性差的三个原因：(1) 不合理的架构采样分布导致了评估偏差。具体地，在超网络的训练过程中，搜索空间中的所有架构有相同的概率被采样训练。一方面，由于参数量较小的架构更容易被训练充分，在相同的采样概率下其

图 7.31 NB201 上 P@topK 和 P@bottomK 的对比^[236]

评估性能往往比参数量大的架构更高，但通常来说，参数量大的架构实际性能会更高，训练不充分会导致它们被低估；另一方面，结构简单的架构通常有更多同质化架构，这些同质化的架构在训练时有着非常相似的梯度更新方向，这就导致超网络的权重更新方向偏向于简单架构的方向，从而高估了简单的架构；(2) 随机的采样和训练策略导致了评估波动。一种典型的误差现象是多模型遗忘现象，即后被采样训练的架构权重会覆盖先前被采样训练的架构在超网络上的训练权重，从而导致超网络“遗忘”了之前的权重信息。图7.32展示了在三个 NAS 基准集上因遗忘现象导致的性能波动，可以看到，这一波动在超网络训练前中期尤为显著。此外，使用不同的随机种子训练，超网络评估结果会有明显差异，这同样说明随机的采样和训练策略对评估结果带来的波动；(3) 造成评估偏差和评估波动的本质原因是权重共享策略共享了大量架构的权重，而这些架构在独立训练时往往会得到差异很大的权重。

图 7.32 权重共享策略导致的多模型遗忘现象^[236]：X 轴代表训练轮次，Y 轴代表平均遗忘值。每一轮中若干个架构会被采样做训练，定义其中单个架构的遗忘值为，该轮次内所有架构训练完之后该架构的性能与该架构刚被训练完后的性能之差。对每一轮中所有被采样架构的遗忘值求平均得到该轮次的平均遗忘值。平均遗忘值距离 0 越远，说明遗忘现象越明显。

改进方向和相关研究

针对上一小节分析得到的两方面原因，Ning 等人给出了三个改进权重共享评估策略的可行思路：(1) 针对评估偏差，设计更公平的架构采样策略，使得所有架构能以更合理的概率分布被采样和训练，缓解架构被高估或低估的现象；(2) 针对评估波动，采用集成的方式提高评估结果的稳定性，例如将训练超网络过程中不同轮次的结果进行集成，来减小随机种子带来的评估误差；(3) 针对大量架构权重被共享导致的评估偏差和波动，适当调整超网络的权重共享程度，例如通过减小搜索空间的规模来减小共享程度，从而缓解多模型遗忘现象；

最新的研究工作大多关注于通过调整超网络的权重共享程度来改善权重共享策略的评估性能。本小节以 Zhou 等人的工作 CLOSE^[390] 为例，展示一个具体有效的改进方案。该工作在多个 NAS 基准集上获得了 SOTA 的评估相关性结果。

不同于以往研究认为较大的权重共享程度会导致评估性能不准确的问题，作者通过实验分析发现，适当地增大共享程度，能在训练前期明显地提升超网络的评估质量。如图7.33所示，Supernet-2 比 Supernet-1 具有更大的共享程度，其评估结果的 KD 系数在训练前期更高。基于此，作者提出设计一种课程学习式的调整方案 CLOSE，在超网络的训练过程中动态调整超网络的权重共享程度，即在训练前期使用更大的共享程度，使得评估相关性迅速增长，在训练后期逐渐降低共享程度，使训练收敛时的评估相关性获得提升。

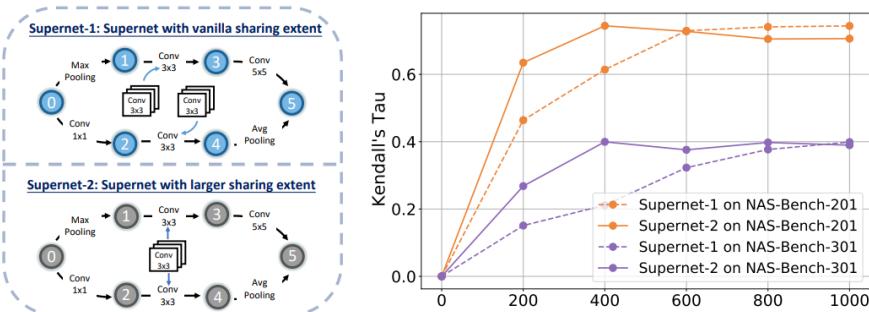


图 7.33 增大权重共享程度对评估相关性的影响^[390]

此外，作者还发现传统的权重共享模式存在问题。例如，如图7.34中所示的两个模型架构 (a) 和 (b)，它们是一对同质化的架构，在相同的训练环境下对应位置的运算符应当获得完全相同的权重，即 (a) 架构中 0 到 2 号点之间的 1×1 卷积和 (b) 架构中 0 到 1 号点之间的 1×1 卷积权重相同。但传统的共享模式并不共享这两个运算符。为

解决该问题，作者设计了一种新型的超网络 CLOSENet。CLOSENet 将超网络的权重存储和权重分配解耦合，并设计了一个基于图架构编码器 GATES^[238]（详细介绍见... 节）的推理模块，来为架构中的操作符合理地分配共享权重。该推理模块能根据架构中每个运算符的架构上下文信息来决定权重的分配模式，通过理论和实验证明，该模块能使具有完全对称或相似的架构上下文的运算符具有相同或相似的权重。

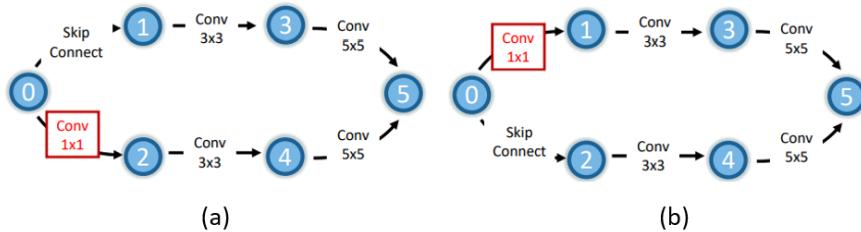


图 7.34 CLOSENet 按照功能信息共享操作符的权重^[390]

7.5.3 基于权重生成的单次评估策略

权重生成超网络 (HyperNet) 最早由 Ha 等人提出^[101]，其思想是利用一个网络（即超网络）生成另一个网络的权重。如图7.35所示，该策略构建了一个超网络 (HyperNet)，该超网络的输入为一个架构，输出为该架构各个位置的权重。基于权重生成的单次评估策略相对于权重共享策略来说应用较少，可能的原因是该类策略对搜索空间的支持不灵活。导致超网络的训练不稳定、难收敛，已有工作的性能较差。

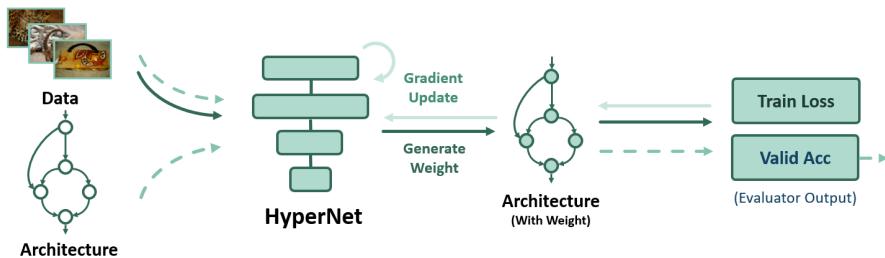


图 7.35 基于权重生成的单次评估策略

Brock 等人于 2017 年首次将权重生成超网络引入到神经网络架构搜索算法中，并基于此提出了 SMASH 算法^[24]，实现高效的架构评估和搜索。SMASH 算法预先训练好一个权重生成超网络，在搜索的过程中，将待评估网络架构的表征输入超网络中，

得到该网络的权重。评估架构性能时，直接利用生成的架构权重进行测试。SMASH 算法对网络架构的表征借鉴了存储器 (Memory Bank) 的工作过程，如图7.36所示，但该方法仅能建模架构的一部分参数，同时难以对具有复杂拓扑结构的网络架构难以合理表征。

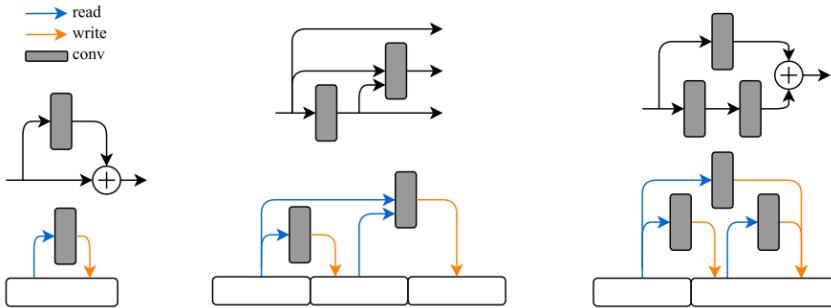


图 7.36 SMASH 算法的架构表征方式^[24]

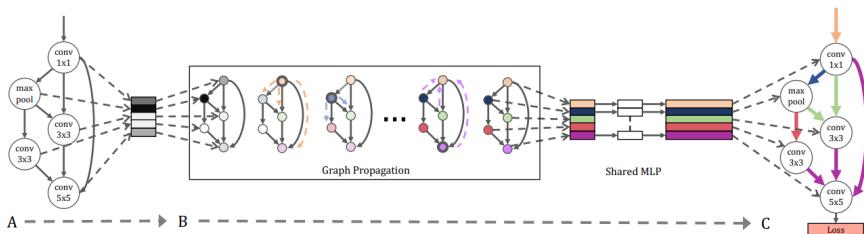


图 7.37 图超网络 (GHN) 的工作流程^[366]

为了得到更合理的架构表征，Zhang 等人提出了基于图卷积神经网络的图超网络 (Graph Hypernetwork, GHN)^[366]，并利用 GHN 进行网络架构的评估和搜索。如图7.37所示，对于随机采样或待评估的网络架构，GHN 利用多层次图卷积神经网络进行前向计算得到架构的表征，随后将架构表征输入一个共享权重的多层感知机 (MLP) 中进行权重预测。

7.5.4 零次评估策略

零次评估策略时目前评估速度最快的一种方式，其不需要依赖训练，至多仅需要花费神经网络进行一次前向计算和梯度反传的时间开销。如图7.38所示，对于给定的架构和任务数据集，先对架构的权重进行随机初始化，然后将一部分数据在模型架构

上做一次前向推理和一次梯度反传，在一些方法中甚至仅需要进行单次前向推理，最终根据某些中间结果计算评估指标，例如对架构中所有可训练权重的梯度求和。目前主流的零次评估指标可以分为下面两类。

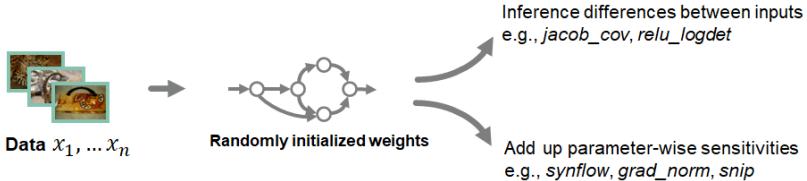


图 7.38 零次评估策略流程

一类主流的零次评估指标是基于模型剪枝领域的权重敏感度设计的^[3]，我们将这类指标称为**参数级别 (Parameter-level)** 的零次评估指标。将模型权重，计算特征和损失函数值分别记作 θ, z 和 \mathcal{L} ，则这些指标的公式可以写为：

$$\begin{aligned}
 plain : \mathcal{S}(\theta) &= \frac{\partial \mathcal{L}}{\partial \theta} \odot \theta; & snip : \mathcal{S}(\theta) &= \left| \frac{\partial \mathcal{L}}{\partial \theta} \odot \theta \right|; & grasp : \mathcal{S}(\theta) &= -\left(H \frac{\partial \mathcal{L}}{\partial \theta} \right) \odot \theta; \\
 synflow : \mathcal{R} &= \mathbb{1}^T \left(\prod_{\theta_i \in \theta} |\theta_i| \right) \mathbb{1}, \mathcal{S}(\theta) & = \frac{\partial \mathcal{R}}{\partial \theta} \odot \theta; & fisher : \mathcal{S}(z) &= \sum_{z_i \in z} \left(\frac{\partial \mathcal{L}}{\partial z} z \right)^2.
 \end{aligned} \tag{7.6}$$

将架构中所有的权重敏感度求和，作为评估架构性能的零次评估指标。

另一类主流的零次评估指标是基于评估架构对输入数据分辨能力的思想设计的，我们将这类指标称为**架构级别 (Architecture-level)** 的零次评估指标。这类指标的代表是由 Mellor 等人设计的 *jacob_cov*^[222] 和 *relu_logdet*^[223]。其中，*jacob_cov* 的计算公式如式 7.7 所示：

$$S = - \sum_{i=1}^N [\log(\sigma_{J,i} + k) + (\sigma_{J,i} + k)^{-1}], \tag{7.7}$$

其中 $\sigma_{J,1}, \sigma_{J,2}, \dots, \sigma_{J,N}$ 是协方差矩阵 $\Sigma_J = \text{cov}(J, J)$ 的特征值， $J = (\frac{\partial f_1}{\partial x_1}, \frac{\partial f_2}{\partial x_2}, \dots, \frac{\partial f_N}{\partial x_N})$ 是网络模型对 N 个输入数据 x_1, x_2, \dots, x_N 的雅可比矩阵。*relu_logdet* 通过架构中所有的 ReLU 层对不同输入产生的激活值差异来反映架构的分辨能力，其计算公式如

式7.8所示：

$$s = \log ||K_H||$$

$$\text{where } K_H = \begin{pmatrix} N_A - d_H(c_1, c_1) & \cdots & N_A - d_H(c_1, c_N) \\ \vdots & \ddots & \vdots \\ N_A - d_H(c_N, c_1) & \cdots & N_A - d_H(c_N, c_N) \end{pmatrix}, \quad (7.8)$$

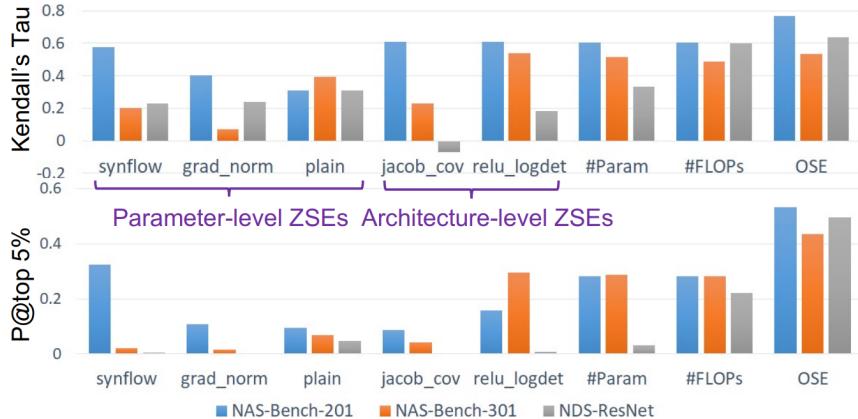
其中 c_i 是一个二进制掩码，表示架构中所有的 ReLU 层对于第 i 个输入数据产生的激活值是否大于零， $d_H(c_i, c_j)$ 表示两个二进制掩码间的汉明距离。

尽管零次评估策略具有非常优秀的评估效率，是目前最快的评估策略，但有研究工作通过实验发现了其评估相关性难以令人满意，仅在计算资源极其受限的场景中才会考虑使用该评估策略。在接下来的部分，本书以 Ning 等人^[236] 的工作为例，介绍零次评估策略具体存在的问题和对应的原因(所涉及到的评估基准集和评估指标介绍见7.5.2节)。

问题与原因分析

Ning 等人通过大量实验观察，总结出了零次评估策略存在严重的评估相关性差的问题，如图7.39所示。同时，作者总结了四个关键现象：(1) 零次评估指标比直接使用架构的参数量 (#Param) 和计算量 (#FLOPs) 作为指标更差；(2) 同一个零次评估指标在不同搜索空间中的表现不同。例如，*jacob_cov* 在 NB101 空间具有较高的相关性性能，但在 NB301 空间却完全失去了架构排序的能力；(3) 在所有搜索空间上，架构级别的零次评估指标相比于参数级别的零次评估指标均获得更高的相关性；(4) 所有的零次评估指标对头部架构的评估准确率 (P@topK) 均很低。

针对上述问题和现象，作者从架构层面和运算符层面分别分析总结了原因：(1) 从架构参数量层面上看，一些零次评估指标（包括 *synflow*、*snip*、*grad_norm*、*fisher* 和 *grasp*）对参数量较大的架构有明显偏好，导致它们的评估性能偏高；(2) 从架构拓扑层面上看，零次评估指标对特定的拓扑结构或运算符有明显偏好，例如，一些参数级别的零次评估指标（包括 *snip*、*grad_norm* 和 *fisher*）偏好于不包含跳连接的线性架构，原因是它们具有很大的梯度，但这类架构通常会产生梯度爆炸的现象，实际的性能较差；而架构级别的零次评估指标（包括 *jacob_cov* 和 *relu_lodget*）更偏好于卷积核尺寸更小的 1×1 卷积，但在实际训练中，包含更多 3×3 卷积的架构往往性能更优。

图 7.39 零次评估策略在 NAS 基准集上的性能对比^[236]

7.6 可微分神经网络架构搜索

2017 年, Liu 等人^[196]首次提出可微分神经网络架构搜索。这类搜索算法均使用共享权重评估策略,但是其架构搜索策略较为特殊 – 基于梯度更新来优化架构决策。具体而言,可微分搜索实现“理解”功能的方式在于为每个待搜索决策分配一组架构权重,并在超网络训练过程中维护并更新所有架构权重。在经典的可微分搜索方法中,架构权重被认为反映了每个候选决策的重要性。其实现“运用”功能的方式是对于每个待搜索决策,选择对应的架构权重值最高的选项作为最终的架构决策。**TODO: 理解与引用可以删掉,反正读者肯定在这里理解不了。**

一方面,可微分搜索基于梯度对模型架构进行优化,相比于离散搜索算法更加高效简洁;另一方面,可微分搜索基于权重共享策略对模型性能进行评估,在搜索过程中协同优化架构与权重,一定程度上增加了模型权重优化与架构优化的适配性。**TODO: 这一段可以删掉**

本节将对可微分神经网络架构搜索进行介绍:以 DARTS^[196]为例,7.6.1节、7.6.2节分别介绍可微分搜索基本的连续松弛方法与优化方法;7.6.3节介绍可微分搜索面临的坍缩问题以及相应的解决尝试;7.6.4节则以 PC-DARTS^[334]、Single-Path NAS^[281]以及 ProxylessNAS^[31]为例介绍高效可微分搜索的尝试。

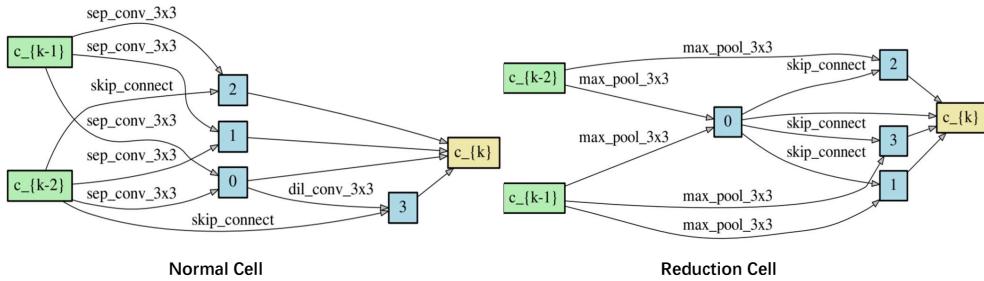


图 7.40 DARTS 搜索得到的卷积单元架构（引自参考文献 [196]）

7.6.1 连续松弛

DARTS 设计了由正常单元与降采样单元堆叠而成的微观架构搜索空间。以其卷积神经网络搜索空间为例，降采样单元位于网络总深度 $1/3$ 和 $2/3$ 的位置，其余为正常单元。图7.40展示了其搜索得到的单元架构。每个单元是由七个节点组成的有向无环图，包含两个输入节点（绿色，来自前序两单元的输出）、四个中间节点（蓝色）以及一个输出节点（黄色）。每个节点 $x^{(i)}$ 是一个特征图，每个有向边 (i, j) 上有一个操作 $o^{(i,j)}$ 、作用在 $x^{(i)}$ 上、结果累加到节点 $x^{(j)}$ 上。每个中间节点的计算过程如7.9式所示。输出节点为四个中间节点的特征在通道维度的拼接。

$$x^{(j)} = \sum_{i < j} o^{(i,j)}(x^{(i)}). \quad (7.9)$$

神经网络架构表征本质是离散的，不能直接应用梯度算法在连续空间进行优化。因此，DARTS^[196] 提出离散的松弛架构决策为连续架构参数表示。具体而言，令 \mathcal{O} 表示候选操作集合，DARTS 里节点 i 到 j 边上的操作是一个“混合操作”，其计算为所有可能候选操作计算的加权融合，如7.10式所示。

$$\bar{o}^{(i,j)}(x) = \sum_{o \in \mathcal{O}} \frac{\exp(\alpha_o^{(i,j)})}{\sum_{o' \in \mathcal{O}} \exp(\alpha_{o'}^{(i,j)})} o(x). \quad (7.10)$$

其中，节点 i 与 j 间不同操作的融合权重由维度为 $|\mathcal{O}|$ 的向量 $\alpha^{(i,j)}$ 表示。用这种松弛方式，架构搜索任务就变成了学习一套连续的变量 $\alpha = \{\alpha^{(i,j)}\}$ 。搜索结束后，通过用最可能的操作替换掉原有的混合操作 $\bar{o}^{(i,j)}$ ，即 $o^{(i,j)} = \operatorname{argmax}_{o \in \mathcal{O}} \alpha_o^{(i,j)}$ ，便可离散得到最终架构。

7.6.2 优化方法

利用松弛方法，任意神经网络模型均可表示为 $\mathcal{A}(\omega, \alpha)$ ，其中 \mathcal{A} 为搜索空间， ω 为模型的权重， α 为模型对应的架构。DARTS 将神经网络架构搜索任务建模为二阶 (bi-level) 优化过程，如7.11式所示。其中 \mathcal{L}_{train} 、 \mathcal{L}_{val} 分别为训练集与验证集上的损失函数，并统一使用梯度算法求解。

$$\begin{aligned} & \min_{\alpha} \mathcal{L}_{val}(\omega^*(\alpha), \alpha), \\ s.t. \quad & \omega^*(\alpha) = \arg \min_{\omega} \mathcal{L}_{train}(\omega, \alpha). \end{aligned} \quad (7.11)$$

然而得到准确的 $\omega^*(\alpha)$ 的内部优化代价极其高昂，DARTS 因此提出一步近似方案近似求解最优权重 $\omega^*(\alpha)$ ，即将模型在训练集上进行一步梯度下降后的权重近似作为当前架构的最优权重，并用此权重对应的验证集损失对架构参数的梯度来更新架构参数 α ，如7.12式所示，其中 ξ 为内部一步优化的学习率。

$$\nabla_{\alpha} \mathcal{L}_{val}(\omega^*(\alpha), \alpha) \approx \nabla_{\alpha} \mathcal{L}_{val}(\omega - \xi \nabla_{\omega} \mathcal{L}_{train}(\omega, \alpha), \alpha). \quad (7.12)$$

应用链式法则化简7.12式，我们能得到

$$\nabla_{\alpha} \mathcal{L}_{val}(\omega', \alpha) - \xi \nabla_{\alpha, \omega}^2 \mathcal{L}_{train}(\omega, \alpha) \nabla_{\omega'} \mathcal{L}_{val}(\omega', \alpha), \quad (7.13)$$

其中 $\omega' = \omega - \xi \nabla_{\omega} \mathcal{L}_{train}(\omega, \alpha)$ 表示一步近似后的模型权重，第二项包含一个代价昂贵的矩阵向量乘积。DARTS 使用有限差分近似降低第二项的计算复杂度。具体而言，记 ϵ 为一个很小的标量， $\omega^{\pm} = \omega \pm \epsilon \nabla_{\omega'} \mathcal{L}_{val}(\omega', \alpha)$ ，有

$$\nabla_{\alpha, \omega}^2 \mathcal{L}_{train}(\omega, \alpha) \nabla_{\omega'} \mathcal{L}_{val}(\omega', \alpha) \approx \frac{\nabla_{\alpha} \mathcal{L}_{train}(\omega^+, \alpha) - \nabla_{\alpha} \mathcal{L}_{train}(\omega^+ - \alpha)}{2\epsilon}. \quad (7.14)$$

求解该有限差分只需要对权重进行两次正向计算，对架构进行两次反向传播，复杂度从 $\mathcal{O}(|\alpha||\omega|)$ 降低到 $\mathcal{O}(|\alpha| + |\omega|)$ 。

整体迭代过程如算法7.4所示，模型收敛时结束搜索。

7.6.3 搜索坍缩

原始的可微分策略在 DARTS 搜索空间中找到的架构可以达到 97.24% 的 CIFAR-10 测试集准确率以及 73.3% 的 ImageNet 测试集准确率。这样的结果在当时的 NAS 研

算法 7.4 DARTS 算法流程

- 1: Create a mixed operation $\bar{o}^{(i,j)}$ parametrized by $\alpha^{(i,j)}$ for each edge (i,j)
 - 2: **while** not converged **do**
 - 3: 1. Update architecture α by descending $\nabla_\alpha \mathcal{L}_{val}(\omega - \xi \nabla_\omega \mathcal{L}_{train}(\omega, \alpha), \alpha)$
 - 4: ($\xi = 0$ if using first-order approximation)
 - 5: 2. Update weights ω by descending $\nabla_\omega \mathcal{L}_{train}(\omega, \alpha)$
 - 6: **end while**
 - 7: **Return** the derived final architecture based on the learned α
-

表 7.3 NAS-Bench-201 搜索结果 (引自参考文献 [333])

Method	CIFAR-10		CIFAR-100		ImageNet-16-120	
	validation	test	validation	test	validation	test
DARTS-V1	39.77±0.00	54.30±0.00	15.03±0.00	15.61±0.00	16.43±0.00	16.32±0.00
DARTS-V2	39.77±0.00	54.30±0.00	15.03±0.00	15.61±0.00	16.43±0.00	16.32±0.00
SETN	82.25±5.17	86.19±4.63	56.86±7.59	56.87±7.77	32.54±3.63	31.90±4.07
ENAS-V2	39.77±0.00	54.30±0.00	15.03±0.00	15.61±0.00	16.43±0.00	16.32±0.00
Random Sample	90.03±0.36	93.70±0.36	70.93±1.09	71.04±1.07	44.45±1.10	44.57±1.25
NPENAS	91.08±0.11	91.52±0.16	-	-	-	-
REA	91.19±0.31	93.92±0.30	71.81±1.12	71.84±0.99	45.15±0.89	45.54±1.03
NASBOT	-	93.64±0.23	-	71.38±0.82	-	45.88±0.37
REINFORCE	91.09±0.37	93.85±0.37	71.61±1.12	71.71±1.09	45.05±1.02	45.24±1.18
Optimal	91.61	94.37	73.49	73.51	46.77	47.31
ResNet	90.83	93.97	70.42	70.86	44.53	43.63

究中是非常优秀的。然而，可微分搜索策略在某些搜索空间上的表现差强人意，例如 NAS-Bench-1Shot1^[362]、NAS-Bench-201^[67]等。表7.3比较了DARTS与其他搜索策略在 NAS-Bench-201 搜索空间中得到的架构的准确率。前者的性能甚至被随机采样超越，比如其得到的架构在 CIFAR-10 数据集上的测试集准确率仅有 54.30%，显著劣于随机采样到的架构（93.70%）。

可微分搜索存在严重的坍缩现象。图7.41形象化地展示了该现象，其中横轴为搜索 epoch，左侧纵轴（蓝线）为搜索得到的架构从头训练后在测试集上的性能。当搜索进行到某个阶段，虽然超网络的验证损失始终在下降，但是搜索得到的架构的性能却会突然变差。特别地，Liang 等人^[183]发现随着搜索的进行，DARTS 越来越倾向于选择跳跃连接（skip connection）操作（右侧纵轴绿线表示搜索得到的架构单元中跳跃连接的数目）。然而，数量过多的跳跃连接对于架构性能是有害的。因此，搜索得到的架构的测试集性能在后期随着架构中跳跃连接的进一步增多而大幅下降。

关于坍缩现象产生的原因众说纷纭，以下介绍几种经典的假说与解决方式。

权重参数与架构参数之间的优化竞争导致后者优化不充分。Liang 等人^[183]认为坍缩现象源于模型权重与架构在二阶交替优化过程中的合作与竞争。在搜索早期，架构与权重协同优化，一起使搜索到的模型的性能上升。但是由于模型权重的数量远大于

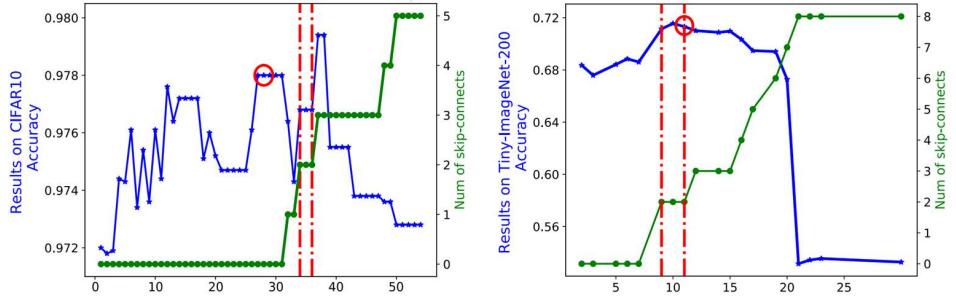


图 7.41 DARTS 算法坍缩问题（引自参考文献 [183]）

架构参数的数量，搜索会逐步倾向于优化权重参数。特别地，搜索空间由几类单元堆叠而成，每类单元共享相同的架构。浅层单元由于靠近整个模型的输入，输入特征更加稳定，而底层单元的输入特征包含更多噪声。因此模型倾向于在浅层单元完成相关计算，而在后层单元利用更多的跳跃连接保留前层计算的特征。由于浅层与后层的某些单元类别相同，因此这类单元都会更多地选择跳跃连接，导致整个模型中的跳跃连接数目增多。他们的解决方式十分简单，即在正常单元中的跳跃连接数目达到或超过 2 时停止搜索。

超网络的优化过程对于某些操作存在偏见。超网络的训练过程与架构的独立训练过程不可避免地存在差距，前者在优化过程中对于某些操作存在偏见。容量较小的架构往往能得到更加充分的优化（7.5.2节），因此跳跃连接等容量较小的操作易被高估。Chu 等人^[55]还指出跳跃连接本身具有相较于其他操作不公平的优化优势。他们认为跳跃连接不仅是构建神经网络模型的一种算子，也是辅助模型优化的一种技巧。其存在缓解了梯度消失问题，因此模型在优化时会倾向于选择跳跃连接辅助优化，但是这对于模型本身的表征能力是不利的。为此，他们提出在搜索过程中引入辅助跳跃连接来去除这种不公平优势。

架构松弛存在性能差距。如式7.10所示，经典的可微分搜索策略以架构权重 α 经过 softmax 归一化后的连续形式表示架构。然而，最终的架构决策却是离散的 0/1 表示。从连续到离散的架构转换带来了较大的性能差距。

TODO：“架构松弛存在性能差距”表述修改。补充一下逻辑。

一方面，架构权重经过归一化后的值往往距离 0/1 较远，而直觉上越远的距离带来的性能差距越大。为鼓励归一化后的值接近 0/1，Chu 等人^[56]在 FairDARTS 中为优化架构权重专门设计损失函数。与此同时，他们还发现不同候选决策归一化后的值的差异较小，因此转而使用 sigmoid 函数以打破不同候选操作之间的直接竞争。此外，经典的可微分策略在做出架构决策时仅保留对应架构权重值最大的操作，使得不同的架

表 7.4 可微分搜索坍缩现象经典假说与解决方式总结

假说	解决方法
超网权重与架构参数存在优化竞争	控制单元中的跳跃连接数目 ^[183]
超网优化对某些操作存在偏见	在搜索过程中引入辅助跳跃连接 ^[55]
架构松弛带来性能差距	使用 sigmoid 函数替代 softmax 函数 ^[56] 设计损失函数使归一化后的权重值接近 0/1 ^[56] 使用 Gumbel softmax 重参数化技巧 ^[66, 320] 以海森矩阵值作为评价指标进行搜索早停 ^[361] 引入噪声平滑损失函数 ^[43] 用去除操作带来的性能损失作为重要性度量 ^[307]

构权重值 (如 $[0.1, 0.3, 0.6]$ 与 $[0.1, 0.4, 0.5]$) 之间可能没有本质区别。因此，一些研究^[66, 320] 利用 Gumbel softmax 以可微分的方式将采样概念引入可微分搜索，将归一化后的架构权重值视为不同操作被选择的概率，并通过选择合适的温度系数调整分布的平滑度。

另一方面，验证损失函数对架构参数 α 并不平滑。Zela 等人^[361] 发现搜索得到的架构在测试集上的性能随搜索过程不断下降，但是在验证集上的性能却不断上升，由此认为 DARTS 搜索到的架构泛化性差是坍缩现象的原因之一。他们发现海森矩阵值 $\nabla_{\alpha}^2 \mathcal{L}_{valid}$ 随搜索过程不断变大。换言之，验证损失函数对架构参数 α 并不平滑，因此离散得到最终架构的过程引入了不可忽视的性能差距。基于以上实验观察，他们提出以海森矩阵值作为评价指标进行早停。从该角度出发，Chen 等人^[43] 在 SmoothDARTS 中提出在架构参数 α 中引入噪声来平滑损失函数，Wang 等人^[307] 则在 DARTS+PT 中提出用去除某操作带来的性能损失度量该操作的重要性。

表7.4简单地总结了以上的经典假说以及相应的解决方式。

7.6.4 更高效的可微分架构搜索

相比于过去的神经网络架构搜索算法，可微分搜索大幅提升了搜索效率，但是由于需要对所有操作进行更新与计算，其所需的内存与计算开销非常大。本小节以 PC-DARTS、Single-path NAS 与 ProxylessNAS 为例讲解当前研究降低可微分搜索开销的尝试。

Partially-Connected DARTS (PC-DARTS) ^[334] 的核心思想是在通道的子集中执行操作搜索，这基于在通道子集上进行操作搜索是在全部通道上进行操作搜索的良好代理的假设。具体而言，如图7.42所示。以 x_i 到 x_j 的连接为例，定义一个将 0 与 1 分

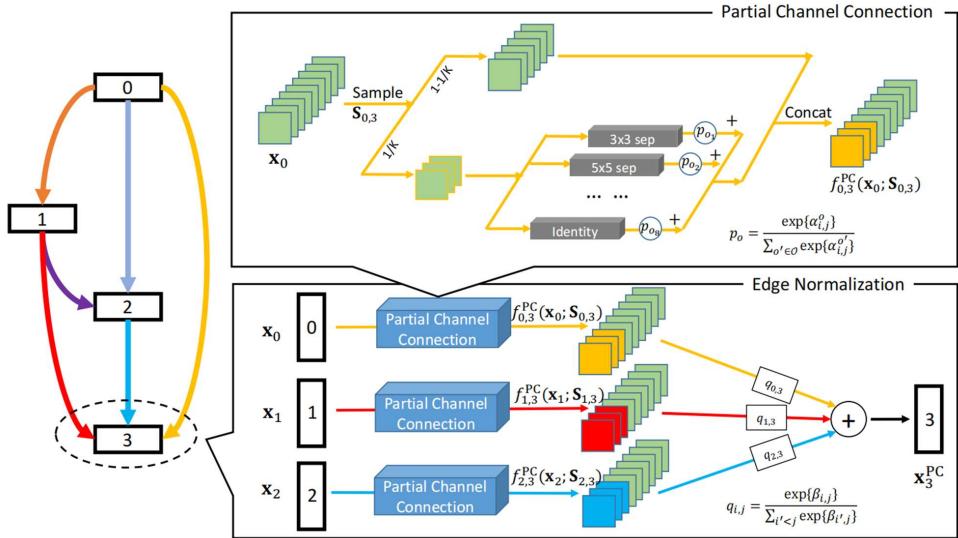


图 7.42 PC-DARTS 算法

别分配给选定与屏蔽的通道的采样掩码 $S_{i,j}$ 。被选定的通道被送入混合操作 \bar{o} 中进行计算，而被屏蔽的通道则通过跳跃连接直接进入输出。

$$f_{i,j}^{PC}(x_i; S_{i,j}) = \sum_{o \in \mathcal{O}} \frac{\exp(\alpha_o^{(i,j)})}{\sum_{o' \in \mathcal{O}} \exp(\alpha_{o'}^{(i,j)})} o(S_{i,j} * x) + (1 - S_{i,j}) * x_i. \quad (7.15)$$

在实践中，PC-DARTS 将所选通道的比例设置为 $1/K$ ，其中 K 为超参数。通过改变 K 值，可以在架构搜索的准确性和效率之间进行权衡以达到平衡。

一方面，通道采样极大地减少内存和计算成本，还将正则化引入操作搜索过程，降低了陷入局部最优的可能性；另一方面，由于架构参数是在随机采样的通道上进行计算的，因此计算得到的最优连接可能并不稳定。为了缓解该问题，PC-DARTS 进一步引入边正则化，即给每条边 (i,j) 一个权值 $\beta_{i,j}$ 。此时，节点 x_j 的计算表达式如下：

$$x_j^{PC} = \sum_{i < j} \frac{\exp(\beta_{i,j})}{\sum_{i' < j} \exp(\beta_{i',j})} \cdot f_{i,j}(x_i). \quad (7.16)$$

特别地，搜索结束后，边 (i,j) 的连接同时由 $\{\alpha_o^{(i,j)}\}$ 与 $\beta_{i,j}$ 决定。具体而言，将归一化系数相乘，即 $\frac{\exp(\beta_{i,j})}{\sum_{i' < j} \exp(\beta_{i',j})}$ 乘以 $\frac{\exp(\alpha_o^{(i,j)})}{\sum_{o' \in \mathcal{O}} \exp(\alpha_{o'}^{(i,j)})}$ ，然后像 DARTS 一样通过查找最大边权值来选择边。

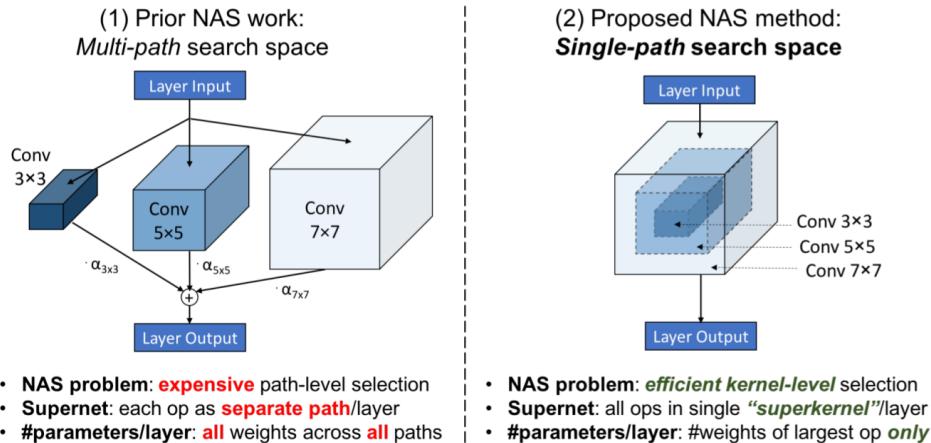


图 7.43 Single-path NAS 与传统 NAS 算法的对比

Single-Path NAS^[28] 则是进一步加大权重共享程度。Stamoulis 等人观察到，只需将外层权重置零，一个大的卷积核就可以表示小的卷积核。因此两节点间不同卷积核大小的候选卷积操作，在超网络中只需用其中最大的卷积核即可表示。具体而言，图 7.43 所示的两节点间包含 3x3 卷积、5x5 卷积与 7x7 卷积三种候选操作。左图展示了传统基于权重共享的神经网络架构搜索算法，在超网络中需要三个独立的卷积核分别代表每种操作，这篇文章称之为多径 NAS 方法。而右图则展示了 Single-Path NAS 提出的超级卷积核 (super kernel)，其本质就是一个 7x7 的卷积核。只需将该超级卷积核最外层的权值置零，就可表示 5x5 的卷积。而将其最外两层权值置零，就可表示 3x3 的卷积。此时超网络只需优化这一个超级卷积核即可。此外，由于跳跃连接本身不需要任何权值，因此该超级卷积核也能表示跳跃连接。这种方法就称之为单路径 NAS 方法。

ProxylessNAS^[31] 的核心思想是通过减少可微分搜索中超网络前馈过程需要保留的候选路径数目来降低 GPU 内存开销。具体来说，原始的可微分搜索基于式 7.10 定义的混合操作进行前馈计算，超网络训练内存开销随候选操作数目线性增长。具体而言，若记候选操作数目为 N ，超网络训练开销大致相当于训练单个模型开销的 N 倍。这使得原始的可微分搜索往往需要在替代任务（如较小的数据集、较小的模型）上进行搜索，再迁移到目标任务。然而替代任务与目标任务间不可避免地存在差距，会对搜索结果产生不良影响。

ProxylessNAS 提出应用二值化门以降低架构搜索的内存开销。与原始的可微分搜索相同，ProxylessNAS 交替训练模型权重与架构参数。其中权重训练过程如图 7.44(1)

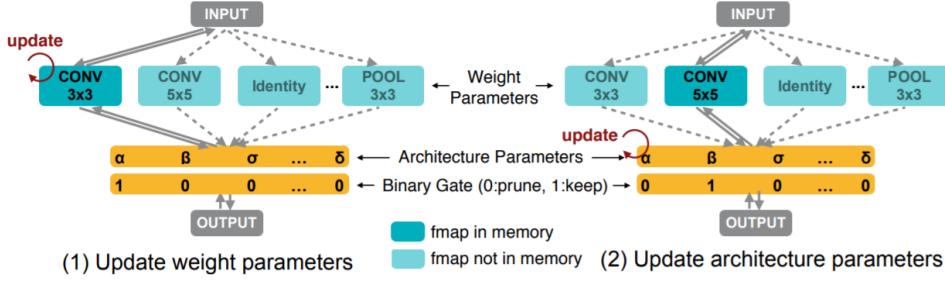


图 7.44 ProxylessNAS：学习权重参数与架构参数。引自参考文献^[31]

所示。输入节点与输出节点间存在四个候选操作。在前馈计算时，ProxylessNAS 不执行式7.10所示的混合操作计算，而是仅根据二值化门激活并更新一条路径（例如，图中 1 表示激活，0 表示裁剪，前馈计算激活 conv 3x3 操作）。通过这种方式，内存开销可大致降为原方法的 $1/N$ 。

形式化地，二值化门 g 的采样如式7.17所示：

$$g = \text{binary}(p_1, \dots, p_N) = \begin{cases} & \text{with probability } p_1 \\ & \dots \\ [0, 0, \dots, 1] & \text{with probability } p_N \end{cases} \quad (7.17)$$

概率 p_i （路径权值）的计算如式7.18所示，其中 $\alpha = \{\alpha_1, \dots, \alpha_N\}$ 是为该待搜索操作分配的架构参数。

$$p_i = \frac{\exp(\alpha_i)}{\sum_j \exp(\alpha_j)}. \quad (7.18)$$

此时式7.10定义的混合操作转化为式7.25所示形式，其中 $\mathcal{O} = \{o_1, \dots, o_N\}$ 。

$$m_{\mathcal{O}}^{\text{Binary}}(x) = \sum_{i=1}^N g_i o_i(x) = \begin{cases} o_1(x) & \text{with probability } p_1 \\ \dots \\ o_N(x) & \text{with probability } p_N \end{cases} \quad (7.19)$$

权重训练结束后，ProxylessNAS 暂时冻结模型权重，并根据式7.17重置二值化门，接着执行如图7.44(2) 所示的架构参数训练过程。然而，ProxylessNAS 中的架构参数并未直接出现在计算图中，无法直接执行梯度优化，因此提出使用式7.20进行近似优化。

$$\frac{\partial L}{\partial \alpha_i} = \sum_{j=1}^N \frac{\partial L}{\partial p_i} \frac{\partial p_i}{\partial \alpha_i} \approx \sum_{j=1}^N \frac{\partial L}{\partial g_i} \frac{\partial p_i}{\partial \alpha_i} = \sum_{j=1}^N \frac{\partial L}{\partial g_i} \frac{\partial(\frac{\exp(\alpha_j)}{\sum_k \exp(\alpha_k)})}{\partial \alpha_i} = \sum_{j=1}^N \frac{\partial L}{\partial g_i} p_i (\delta_{ij} - p_i). \quad (7.20)$$

其中， $\delta_{ij} = (i == j)$ 。如式7.25所示，二值化门 g 在计算图中， $\frac{\partial L}{\partial g_j}$ 可以通过反向传播计算，因此上式可导。但是计算 $\frac{\partial L}{\partial g_j}$ 需要存储 $o_j(x)$ ，上式的内存开销仍然是训练单个模型的 N 倍。

为解决该问题，考虑将该 N 元选择问题分解为多个二元选择任务。ProxylessNAS 每次从 N 个候选路径中基于多项分布 (p_1, \dots, p_N) 随机采样两个路径，然后遮蔽掉其他路径。此时，候选路径数量暂时从 N 减少到 2，并相应重新计算 p_i 与 g_i 。根据式7.20优化被采样的两路径的架构参数 α_i 。最后，由于路径权值是通过对架构参数使用 softmax 来计算的，通过乘以一个比率来缩放这两个被更新的架构参数，以保持未采样路径的路径权值不变。综上，无论 N 的值是多少，每次架构参数优化仅涉及两条路径，从而将内存需求降低到与训练复杂模型相同的水平。

7.7 考虑硬件效率的搜索

考虑到神经网络部署在硬件设备上的资源限制，模型必须兼顾任务性能与硬件效率。因此，面向实际应用的 NAS 往往需要同时对任务性能与硬件效率进行建模与优化，称之为考虑硬件效率的神经架构搜索（Hardware-aware NAS）。

如前文所述，NAS 由搜索空间、评估策略与搜索策略三个组件构成。与之对应，考虑硬件的 NAS 有以下三个主要研究问题：1) **考虑硬件效率的搜索空间设计**。考虑硬件效率的 NAS 需要设计硬件友好的搜索空间。如7.3.1节所述，宏观搜索空间规模过大，微观搜索空间则因层间多样性的匮乏与拓扑结构的复杂不利于移动端的部署。因此，研究者主要通过设计非拓扑搜索空间^[320] 与层次化搜索空间^[296] 提升模型的任务性能与硬件效率；2) **硬件效率加速评估**。在考虑硬件效率的 NAS 中，硬件指标（如延迟、峰值能耗）的获取是架构评估的重要组成部分。传统方法^[186, 296, 349] 将模型实际部署在目标硬件设备上进行测试，但是成本极大。因此，研究者转向使用代理指标^[121, 389]、基于查找表的线性相加^[30, 144, 320]、白盒仿真^[145, 218]、黑盒预测^[70, 238] 等方法高效地评估候选架构的硬件效率；3) **考虑多种硬件效率目标的搜索策略**。常规的 NAS 往往仅针对单一的任务性能指标（如分类任务的准确率）进行搜索，而考虑硬件效率的 NAS 还需要兼顾多种硬件指标。而且这些指标相互之间往往存在权衡，例如任务性能与延迟往往不能同时达到最优。研究者将该条件下的搜索定义为约束优化问题或者多目标优化

问题，并提出了标量化^[296]、帕累托搜索^[71, 122]等方法进行求解。

此外，实际应用场景中存在各种各样的硬件设备。如果针对每种硬件设备及相应的约束都需要从头进行搜索，将带来巨大的计算开销。因此，OFA^[30]等工作提出面向多种硬件设备及约束的 NAS 方法，高效地为多种硬件及约束提供架构设计。

本节将围绕上述四个研究问题进行展开。7.7.1节将以 MNASNET^[296]为例介绍搜索空间设计；7.7.2节将介绍硬件指标的加速评估方法；7.7.3节将介绍考虑多种目标的搜索策略；7.7.4节将以 OFA 为例介绍面向多种硬件及约束的搜索方法。

7.7.1 考虑硬件效率的搜索空间

如7.3.1节所述，微观架构搜索空间搜索一种可重复的结构单元，其搜索决策仅包括单元架构的算子种类及算子之间的连接方式，并通过堆叠搜索得到的单元架构构成整个网络。这类搜索空间因为显著减小了空间大小并增加了网络拓扑的复杂性与多样性，所以得到了广泛的应用。

然而，微观架构搜索空间并不适合考虑硬件效率的 NAS。首先，**层间多样性**^[296, 320]对于考虑硬件效率的架构设计极其重要。同样的算子或者模块在不同的网络层对于延迟等硬件指标的影响不同，因此不同的网络层需要单独进行架构设计。而微观架构搜索空间人为固定了堆叠方式，且每一层的单元架构完全相同，使得搜索得到兼顾任务性能与硬件效率的架构变得极其困难。其次，**硬件高效的神经网络架构不适合太过复杂的拓扑结构**。微观搜索空间单元架构过于复杂的拓扑结构使得架构在硬件上的推理延迟较大，难以满足移动端部署的需求。

针对考虑硬件效率的 NAS，研究者主要设计非拓扑搜索空间与层次化搜索空间。相关内容在7.3.1节进行了介绍。本小节将以 MNASNET^[296]设计的层次化搜索空间为例，从层间多样性角度介绍考虑硬件效率的搜索空间设计。

如图7.45所示，MNASNET 的搜索空间由 7 个预定义的模块（Block）堆叠而成。与绝大多数用于图像分类任务的神经网络类似，从模型的输入端到输出端，这些模块的输入分辨率（input resolution）逐步下降，输出通道数（filter size）逐步上升。每个模块由数目可搜索的层（Layer）堆叠而成。为了控制搜索空间的规模，每个模块中不同层的架构保持一致。例如，Layer 4-1 与 Layer 4-N₄ 的架构一致。

在微观搜索空间中，不同模块的架构往往保持一致。但是为了鼓励层间多样性，**MNASNET 允许不同模块拥有不同的架构**。MNASNET 分别对 7 个模块的架构进行搜索，使其可以根据模块的输入分辨率专门对架构进行设计，从而达到更好的性能与效率间的权衡。具体地，每个模块 i 的搜索空间包含以下内容：1) 层架构的卷积类型（ConvOp）；2) 层架构的卷积尺寸（KernelSize）；3) 层架构的 SE^[124] 系数（SERatio）；

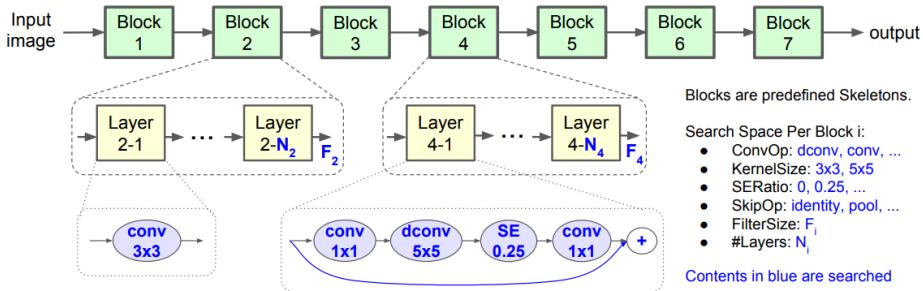


图 7.45 MNASNET 搜索示意图。引自参考文献^[296]

4) 层架构的跳跃连接类型 (SkipOp); 5) 输出通道数 F_i (FilterSize); 6) 层数目 N_i (#Layers)。

在 MNASNET 搜索空间中搜索得到的架构以 78ms 的延迟达到了 75.2% 的 ImageNet 准确率，远优于在 Zoph 等人^[393]设计的微观搜索空间中得到的架构性能 (100ms, 72.0%)。此外，在消融实验中，MNASNET 将层间多样性空间的搜索结果与非层间多样性（保持所有 block 的架构相同）空间的搜索结果进行了对比。在相似的延迟下，前述在层次多样性空间中得到的架构，远优于在非层次多样性空间中得到的架构 (79ms, 72.5%)。这体现了搜索空间的层次多样性对于考虑硬件效率的 NAS 的重要性。此外，7.3.1 节介绍的 FBNet^[320] 非拓扑搜索空间也是考虑硬件效率的 NAS 的经典设计，其实验结果充分体现了层间多样性与拓扑简单性的意义。

7.7.2 硬件效率加速评估

如何高效准确地获得神经网络在硬件设备上的效率指标是考虑硬件效率的 NAS 的关键问题之一。最朴素的方法^[186, 296, 349]将候选模型实际部署在目标硬件上进行测试，以此获得最为准确、真实的硬件性能。然而，将候选模型部署到目标硬件上进行测试的流程往往十分繁琐，且有时并不可行。因此，研究者诉诸其他方法对硬件效率进行加速评估。本小节将分别对这些方法进行介绍。

一、使用代理指标

这类方法^[121, 389] 使用参数量、FLOPs、MACs 等易于获得的指标作为硬件效率的代理度量。例如，用 FLOPs 代理延迟，用参数量代理内存开销。由于这些代理指标的获取开销往往可以忽略不计，因此这类方法十分高效。然而，既有研究^[70, 216]指出代理指

标往往不能真实地反映硬件效率。例如，FLOPs 较小的架构并不一定具有更高的计算效率。

二、使用查找表进行线性相加

这类方法^[30, 144, 320] 将神经网络视为一个个基本模块的组合，假设模型的计算过程就是组成它的基本模块依次计算的过程，将基本模块的硬件指标之和作为整体模型的硬件指标。例如，FBNet 将每个基本模块的延迟相加作为整体模型的延迟估计。具体而言，这类方法首先会建立一张基本模块的硬件指标查找表，然后依次遍历模型中的基本模块并累加基本模块在查找表中对应的硬件指标。然而，该类方法的误差较大^[70]。在有着并行计算/数据搬运能力的硬件平台上（例如 CPU 与 GPU），由于硬件平台的调度策略，每个基本模块的硬件指标总和可能会显著偏离整体模型的硬件指标。

三、白盒仿真

在对硬件的计算架构及运算资源全面了解的前提下，研究者可以构建对应的计算模型，模拟神经网络在硬件设备上的计算过程，从而估算候选架构的硬件指标^[145, 218]。例如，FNAS^[145] 将运算任务分割为若干个子任务，并建立输入数据-运算单元-输出数据的任务依赖图。其中，每个子任务所需计算时间可以直接根据公式计算而得。根据任务依赖图、数据复用策略以及子任务所需的时间，FNAS 可以直接计算得到整个任务所需的计算时间。

四、黑盒预测

黑盒方法^[145, 218] 隐式地对神经网络与目标硬件间的关系进行建模，从而获取候选架构在硬件设备上的预期性能。这类方法通过拟合一定量的架构-硬件性能数据，获得对未知架构的硬件性能的估计。与 7.4.4 介绍的架构性能预测器类似，这些黑盒模型通常也以架构为输入、硬件指标为输出，由架构编码器与预测头两部分构成。

本小节以 BRP-NAS^[70] 为例介绍黑盒预测器的构建方法。BRP-NAS 应用图卷积神经网络（Graph Convolutional Network, GCN）对架构进行编码，使用黑盒预测器对 NAS-Bench-201 中架构的延迟进行快速评估。该 GCN 包含四个由 600 个隐藏单元构成的网络层，其输出的架构隐表征通过一个全连接层映射到最终的延迟预测值。

具体而言，记 NAS-Bench-201 搜索空间单元架构的节点个数为 N，每个节点的特征用一个长度为 D 的独热编码向量表示，则所有节点的特征可以构成 $N \times D$ 的特

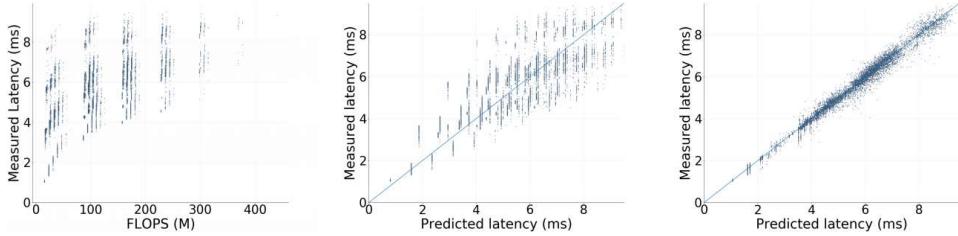


图 7.46 从左至右：使用代理指标 FLOPs / 基于查找表的线性相加方法 / GCN 编码的黑盒模型获得的延迟评估与真实延迟间的散点图。引自参考文献^[70]

征矩阵 X 。与此同时，单元架构中节点连接关系可以表示为 $N \times N$ 的邻接矩阵 A 。该 GCN 第 l 层的前向计算关系如下：

$$H^{l+1} = f(H^l, A) = \sigma(AH^lW^l), \quad (7.21)$$

其中， H^l 与 W^l 分别表示第 l 层的输入特征与权重， σ 为激活函数， $H^0 = X$ 。

BRP-NAS 将该延迟预测器在 NAS-Bench-201 搜索空间中随机采样的 900 个架构上进行训练，用于预测非训练集中架构的延迟。图 7.46 从左至右分别展示了使用代理指标 FLOPs、基于查找表的线性相加、基于黑盒模型三种方法获得的延迟评估与真实延迟间的散点图。从左图可以看到，FLOPs 并不是一个好的延迟代理指标。对比右面两图则可以看到，基于黑盒模型得到的延迟预测值明显比基于查找表的线性相加方法得到的延迟预测值更加准确。

7.7.3 考虑多种硬件效率目标的搜索策略

常规的 NAS 往往仅针对单一的任务性能指标进行搜索。在搜索过程中，搜索策略以该单一的任务性能为奖励进行采样。然而，在考虑硬件效率的 NAS 中，搜索算法不仅需要考虑任务性能，还需要考虑一种或多种硬件效率指标。并且这些指标相互之间往往存在权衡，难以同时达到最优。例如，任务性能较高的架构可能有着更高的计算延迟。

如何在搜索过程中考虑多种目标，是考虑硬件效率的 NAS 的重要研究问题。既有方法将考虑硬件效率的 NAS 定义为约束优化问题或者多目标优化问题，以下将分别进行介绍。

约束优化问题

考虑硬件效率的 NAS 可以被定义为约束优化问题^[17, 31, 296]。在此定义下，架构搜索旨在求解以下方程：

TODO: 公式整体

$$\min_{a \in \mathcal{A}} f(a) \text{ subject to } h_i(a) \leq c_i \text{ for } i \in \{1, \dots, k\} \quad (7.22)$$

其中 a 表示候选架构， \mathcal{A} 表示搜索空间， $f(a)$ 表示任务本身的优化目标（如分类任务的错误率）， $h_i(a)$ 表示第 i 种硬件指标， c_i 表示第 i 种硬件指标的约束。换言之，搜索算法在任何一种硬件指标均满足约束的情况下，最大化架构的任务性能。

为了解决上述优化问题，已有方法大致可分为两类：1) 搜索空间裁剪。这类方法在仅采样满足所有资源约束的架构的前提下最大化任务性能，相当于对搜索空间进行了裁剪。例如，OFA^[30] 在搜索过程中仅采样满足延迟约束的架构；2) 标量化。这类方法人工设计计算规则，将多种搜索目标计算为单一的标量作为奖励。其计算规则往往会对不满足约束的指标进行惩罚。

本小节以 MNASNET 为例，介绍标量化的具体方法。给定架构 a ，记其在分类任务上的准确率为 $ACC(a)$ ，在目标硬件上的延迟为 $LAT(a)$ 。MNASNET 将架构允许的最大延迟 T 视为硬约束，并在延迟限制下最大化正确率，如下式所示：

$$\max_{a \in \mathcal{A}} ACC(a) \text{ subject to } LAT(a) \leq T \quad (7.23)$$

MNASNET 通过在奖励中加入延迟项来考虑硬件效率，其定义的优化目标如下：

$$\max_{a \in \mathcal{A}} ACC(a) \times \left[\frac{LAT(a)}{T} \right]^w, \quad (7.24)$$

其中 w 是权重因子，如下式所示：

$$w = \begin{cases} \alpha & \text{if } LAT(a) \leq T \\ \beta, & \text{otherwise} \end{cases} \quad (7.25)$$

其中 α 与 β 是任务特定的常量。一般而言，当采样架构的延迟 $LAT(a)$ 超过约束 T 时，式 7.24 的第二项会在原有准确率的基础上施加惩罚，反之则施加奖励。施加惩罚或奖励的力度通过 α 与 β 进行调节。通过这样的方式，MNASNET 鼓励搜索得到的架构在拥有接近或小于 T 的延迟的同时有着较高的任务准确率。

多目标优化问题

考虑硬件效率的 NAS 也可以被定义为多目标优化问题^[71, 92, 122, 211]。在此定义下，公式7.22中每一种硬件约束指标均被视为单独的优化目标，与任务本身的优化目标一起进行优化，如下式所示：

$$\min_{a \in \mathcal{A}} (f(a), h_1(a), h_2(a), \dots, h_k(a)) \quad (7.26)$$

标量化方法同样可以被用于求解该多目标优化问题。本小节以基于可微分搜索策略的 FBNet 为例。FBNet 希望搜索到的模型有着尽可能高的准确率与尽可能低的延迟。为此，FBNet 在优化超网络的损失函数中同时考虑了分类任务的交叉熵与网络延迟，如式7.27所示：

$$\mathcal{L}(a, w_a) = \mathbf{CE}(a, w_a) \cdot \alpha \log(\mathbf{LAT}(a))^\beta \quad (7.27)$$

其中，第一项 $\mathbf{CE}(a, w_a)$ 表示带有权重 w_a 的架构 a 的交叉熵损失。第二项中的 $\mathbf{LAT}(a)$ 表示架构的延迟。其中，系数 α 控制损失函数的整体大小，系数 β 调节延迟项的大小。FBNet 使用基于查找表的方式来获得架构的延迟 $\mathbf{LAT}(a)$ 。FBNet 首先测量各个操作的耗时，然后对于每个架构 a ，其延迟按照式7.28进行累加计算，其中 $b_{l,i}$ 为第 l 层的第 i 个候选模块， $\theta_{l,i}$ 为第 l 层的第 i 个候选模块被选择的概率。这样不仅可以方便快速的获得架构的延迟，并且延迟对于 θ 可导。

$$\mathbf{LAT}(a) = \sum_l \sum_i \theta_{l,i} \cdot \mathbf{LAT}(b_{l,i}) \quad (7.28)$$

另一种常见的求解方法是寻找一组非支配解。该非支配解集中的任何一个架构均存在优于其他架构的指标，即不存在完全优于其他架构的架构。该非支配解的集合称为帕累托前沿（Pareto front），相应的搜索称之为帕累托搜索（Pareto search）。具体地，现有方法^[71, 92, 122, 211]通常使用多目标进化算法搜索帕累托前沿。多目标进化算法维护神经网络架构的种群，通过进化当前种群来改进从当前种群获得的帕累托前沿。

TODO: 修改帕累托前沿的定义表述。

7.7.4 面向多种硬件设备及约束的 NAS 方法

不同硬件设备对于神经网络架构的偏好不同。即使是同一台硬件设备，在不同的电池条件以及负载下的最优神经网络架构甚至也存在显著差异。如果针对实际应用场景中每种硬件设备及约束都从头进行架构搜索，计算开销将是巨大的。因此，学

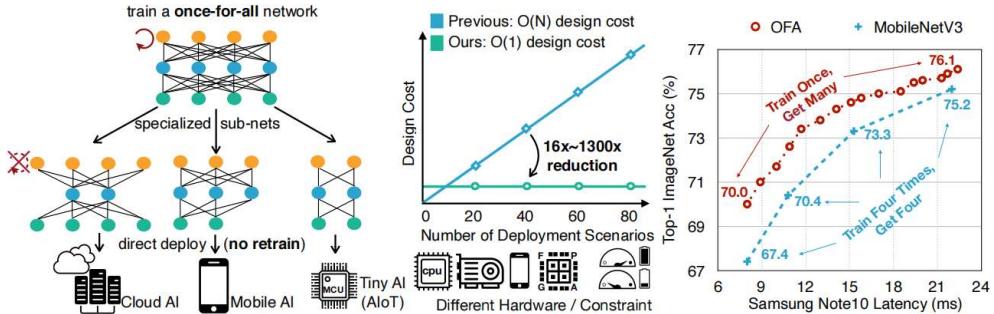


图 7.47 左图：Once-for-all 超网在针对不同硬件的搜索中被用于评估候选架构性能，搜索得到的架构不需要重新训练即可部署；中图：OFA 将搜索复杂度从 $O(N)$ 降至 $O(1)$ ；右图：OFA 可以达到更优异的性能-延迟权衡。

◦

术界与工业界均需要为多种硬件及约束提供架构设计的高效 NAS 方法。本小节将以 OFA^[30] 为例，介绍研究者当前的探索。

OFA 的核心思想 在于将架构训练过程与搜索过程解耦。该方法的核心 在于训练一个Once-for-all 超网络。其作用不在于为不同架构提供准确的性能排序，而在于赋予所有候选架构尽可能高的任务性能。如图7.47（左）所示，OFA 使用候选架构在超网络中的权重进行性能评估以及最终部署，避免了任何后续的训练或微调过程。为进一步提高搜索效率，OFA 基于候选架构在超网络中的性能训练“架构-性能”预测器以快速评估任务性能，并针对目标硬件建立延迟查找表以快速评估延迟。接着，针对该目标硬件及不同约束条件，OFA 即可使用基于进化算法的搜索策略进行快速的架构搜索及最终部署。如图7.47（中）所示，整套搜索流程的复杂度从 $O(N)$ 降至 $O(1)$ ，十分高效。

如图7.48所示，OFA 的搜索空间包含以下四个搜索维度：输入分辨率、卷积核尺寸、网络深度与网络宽度。整个搜索空间包含的候选架构数目高达 2×10^{19} ，并且每种架构还对应着 25 种不同的输入分辨率。如此庞大的搜索空间为 Once-for-all 超网络的训练带来了极大的挑战。如何训练 Once-for-all 超网络？一种可能的方式是将其训练过程视为多目标优化问题，其中每一种目标对应一个候选架构的任务性能。每次迭代中遍历所有候选架构，然后用整体目标的梯度优化超网络。然而，这种训练方法只能应用于候选架构极少的搜索空间。另一种可能的训练方案是在每次迭代中仅采样部分子架构进行更新。然而，作者发现不同子架构在超网络的训练过程中互相干扰，导致任务性能显著下降。

OFA 提出渐进收缩 (Progressive Shrinking) 的超网络训练方式，以提高所有候选架构在超网络中的任务性能。所谓渐进收缩，指的是以渐进的方式执行从容量较大的

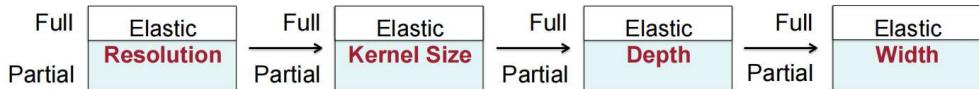


图 7.48 OFA 支持不同深度、宽度、卷积核尺寸以及输入分辨率的渐进收缩过程

表 7.5 硬件感知模型搜索方法小结

方法	搜索策略	硬件效率评估方式	评估策略
MNASNET	基于强化学习	实际部署测量	独立训练
FBNet	可微分搜索	查找表线性相加	基于权重共享的评估
BRP-NAS	基于架构性能预测器	黑盒预测	独立训练
FNAS	基于强化学习	白盒仿真	独立训练
OFA	基于进化算法	查找表线性相加	基于权重共享的评估

子架构到容量较小的子架构的训练顺序。图7.48展示了 OFA 训练超网络过程的示意图。在超网络训练的每次迭代中，OFA 都会采样不同分辨率的图像作为模型的输入，并从采样空间中采样候选架构进行训练，更新相应的超网参数。在超网络训练的起始阶段，采样空间仅包含搜索空间中卷积核最大、深度最深、宽度最大的子架构。随着训练的进行，OFA 逐步将容量较小的子架构加入采样空间，并不断地微调超网络以支持这些新加入的子网络。特别地，OFA 在训练容量较小的子架构时会将容量最大的架构的输出作为软标签进行蒸馏学习。

TODO: 讲一下整体流程

渐进收缩的训练方式在训练容量较小的子架构时，容量较大的子架构已经得到了充分训练。因此，渐进收缩可以防止容量较小的子架构干扰容量较大的子架构的优化。与此同时，渐进收缩训练方式用已经充分训练的容量较大的子架构的权重初始化容量较小的架构，加速了训练流程。而与剪枝相比较，渐进收缩不仅对模型宽度进行调整，还调整了卷积核尺寸、网络深度以及输入分辨率，并且在训练过程中同时微调容量较大与容量较小的子架构。因此，渐进收缩可以训练得到一个性能更强、更加适应不同硬件平台及约束的超网络。如图7.47（右）所示，OFA 可以达到更优异的性能与延迟间的权衡。

在本节的最后，表7.5对前文介绍过的考虑硬件效率的 NAS 方法进行了总结，方便读者进行学习与比较。

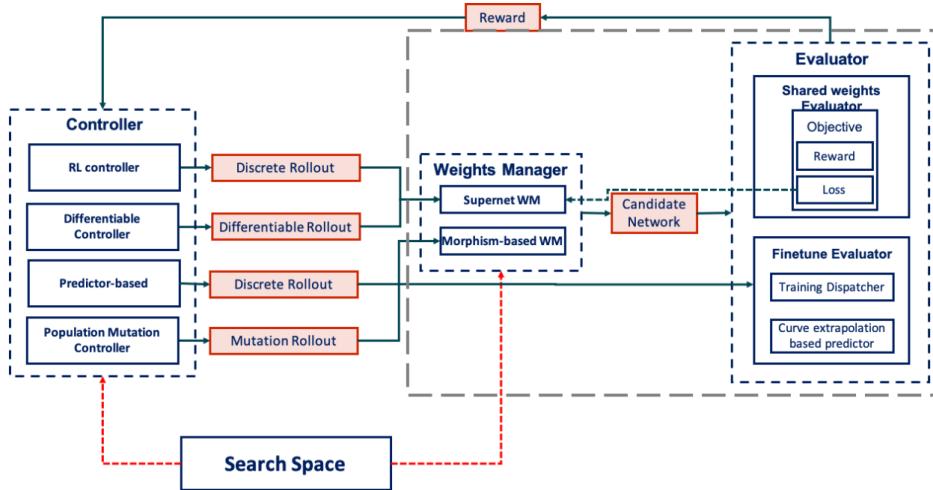


图 7.49 aw_nas 工作流

7.8 实践案例

本节基于清华大学 NICS-EFC 实验室¹与超星未来科技有限公司²联合开发的可拓展集成神经架构搜索框架 aw_nas³，给出 DARTS 算法的实践案例。

7.8.1 aw_nas 概述

开源框架 aw_nas 可用于再现 ENAS、DARTS、OFA 等主流神经架构搜索算法的结果，并应用于图像分类、目标检测、文本建模、硬件容错、对抗鲁棒性等应用与场景。

该框架将神经架构搜索系统划分为以下组件：1) 搜索空间 (search space): 定义包含所有待搜索的神经网络架构的集合；2) 控制器 (controller): 对应搜索策略，定义如何在搜索空间中进行采样；3) 权重管理者 (weights manager): 实现权重共享评估策略，管理超网络的权重；4) 评估器 (evaluator): 对应评估策略，对采样出的候选架构进行性能评估；5) 优化目标 (objective): 定义架构性能的评估指标与模型训练的损失函数。

aw_nas 的工作流如图7.49所示。以可微分搜索为例。首先由可微分控制器 (Differentiable Controller) 从搜索空间中采样架构，并将采样后的架构以可微分 Rollout

¹<http://nicsefc.ee.tsinghua.edu.cn/>

²<https://www.novauto.com.cn/>

³https://github.com/walkerning/aw_nas

(Differentiable Rollout) 的形式传递给超网权重管理者 (Superent WM)。接着，由超网权重管理者将架构与权重新组合成候选网络 (Candidate Network)，传递至评估器 (Evaluator) 进行性能评估。然后，评估器根据定义的优化目标 (objective) 计算该候选架构的损失 (loss) 与性能奖励 (reward)，并对超网权重以及控制器管理的架构参数进行优化。最后，评估器将性能奖励传递至控制器指导下一阶段的采样。

7.8.2 aw_nas 接口

根据代码仓库 README.md 安装框架，然后在命令行输入 `awnas --help` 可以查看框架提供的对外接口。

```

1 07/04 11:41:44 PM plugin      INFO: Check plugins under /home/foxfi/awnas/plugins
2 07/04 11:41:44 PM plugin      INFO: Loaded plugins:
3 Usage: awnas [OPTIONS] COMMAND [ARGS]...
4
5 The awnas NAS framework command-line interface. Use 'AWN_AS_LOG_LEVEL'
6 environment variable to modify the log level.
7
8 Options:
9   --version           Show the version and exit.
10  --local_rank INTEGER the rank of this process [default: -1]
11  --help              Show this message and exit.
12
13 Commands:
14  search             Searching for architecture .
15  mpsearch           Multiprocess searching for architecture .
16  random-sample      Random sample architectures .
17  sample              Sample architectures , pickle loading controller ...
18  eval-arch          Eval architecture from file .
19  derive              Derive architectures .
20  mptrain             Multiprocess final training of architecture .
21  train               Train an architecture .
22  test                Test a final-trained model.
23  gen-sample-config   Dump the sample configuration .
24  gen-final-sample-config Dump the sample configuration for final training .
25  registry            Print registry information .

```

进一步，用户可以查看子命令的用法说明。例如，输入 `awnas search --help` 可以查看 `search` 指令的用法说明。

1	07/04 11:41:44 PM plugin	INFO: Check plugins under /home/foxfi/awnas/plugins
---	--------------------------	-----------------------------------------------------

```

2 07/04 11:41:44 PM plugin           INFO: Loaded plugins:
3 Usage: awnas search [OPTIONS] CFG_FILE
4
5 Searching for architecture .
6
7 Options:
8   --gpu INTEGER          the gpu to run training on [ default : 0]
9   --seed INTEGER          the random seed to run training
10  --load TEXT             the directory to load checkpoint
11  --save-every INTEGER    the number of epochs to save checkpoint every
12  --save-controller-every INTEGER  the number of epochs to save controller checkpoints every
13  --interleave-report-every INTEGER the number of interleave steps to report every, only work in
14                                interleave training mode [ default : 50]
15  --train-dir TEXT        the directory to save checkpoints
16  --vis-dir TEXT          the directory to save tensorboard events. need ‘tensorboard’
17                                extra, ‘pip install aw_nas[tensorboard]’
18  --develop               in develop mode, will copy the ‘aw_nas’ source files into
19                                train_dir for backup [ default : False ]
20  -q, --quiet              [ default : False ]
21  --help                  Show this message and exit.

```

7.8.3 可微分搜索实践案例

使用 aw_nas 实现可微分搜索 DARTS 算法分为搜索（search）与最终训练（train）两个步骤。本小节将分别展开介绍。

aw_nas 通过 YAML⁴文件定义并读取相应的实验配置。仓库根目录下的 YAML 文件 examples/basic/darts.yaml⁵提供了改进版本的可微分搜索配置，其中不同配置的组织如下：1) 基于单元的 CNN 搜索空间；2) CIFAR-10 数据集；3) 可微分控制器；4) 权重共享架构性能评估器；5) 可微分超网络；6) 图像分类优化目标；7) 整个 NAS 搜索流程的编排。

以搜索空间的定义为例。search_space_type 定义使用名称为 cnn 的搜索空间，而 search_space_cfg 则给出相应的超参配置。具体地，在程序运行时，框架会自动识别名称为 cnn 的搜索空间类 CNNSearchSpace，并在实例化搜索空间时传入 search_space_cfg。

```

1 ## ----- Component search_space -----
2 # ----- Type cnn -----
3 search_space_type: cnn

```

⁴<https://yaml.org/>

⁵https://github.com/walkerning/aw_nas/blob/master/examples/basic/darts.yaml

```

4 search_space_cfg:
5   # Schedulable attributes :
6   num_cell_groups: 2
7   num_init_nodes: 2
8   num_layers: 8
9   cell_layout : null
10  reduce_cell_groups:
11    - 1
12  num_steps: 4
13  num_node_inputs: 2
14  concat_op: concat
15  concat_nodes: null
16  loose_end: false
17  shared_primitives:
18    - none
19    - max_pool_3x3
20    - avg_pool_3x3
21    - skip_connect
22    - sep_conv_3x3
23    - sep_conv_5x5
24    - dil_conv_3x3
25    - dil_conv_5x5
26    cell_shared_primitives : null
27  # ----- End Type cnn -----
28 ## ----- End Component search_space -----

```

使用如下指令可以展开搜索

```
1 awnas search examples/basic / darts .yaml --gpu 0 --save-every <SAVE_EVERY> --train-dir <TRAIN_DIR>
```

搜索结束后，实验日志会存放在 <TRAIN_DIR>/search.log 内。日志文件的最后会给出搜索得到的最优架构，例如下方所示的架构。

```
1 CNNGenotype(normal_0=((('sep_conv_3x3', 0, 2), ('sep_conv_3x3', 1, 2), ('skip_connect', 0, 3), ('sep_conv_3x3', 1, 3), ('skip_connect', 0, 4), ('sep_conv_3x3', 1, 4), ('skip_connect', 0, 5), ('skip_connect', 1, 5)), reduce_1=((('max_pool_3x3', 0, 2), ('max_pool_3x3', 1, 2), ('max_pool_3x3', 0, 3), ('skip_connect', 2, 3), ('skip_connect', 2, 4), ('avg_pool_3x3', 0, 4), ('skip_connect', 2, 5), ('avg_pool_3x3', 0, 5)), normal_0_concat=(2, 3, 4, 5), reduce_1_concat=(2, 3, 4, 5)))
```

搜索结束后，使用以上搜索得到的架构替换掉最终训练配置文件 examples/mloss/-

darts/darts_final.yaml⁶内 final_model_cfg 原定义 genotypes。接着使用如下指令可以展开最终从头训练。

```
1 awnas train examples/mloss/ darts / darts_final .yaml --gpu 0 --save-every <SAVE_EVERY> --train-dir <  
TRAIN_DIR>
```

7.8.4 其它

aw_nas 是一款可拓展性极强的神经架构搜索框架，支持插件机制以及基于已有类别的新组件开发。其中插件机制请有兴趣的读者参考 README.md 中的 Plugin mechanism，新组件开发可参考说明 doc/development.md⁷。

7.9 小结

⁶https://github.com/walkerning/aw_nas/blob/master/examples/mloss/darts/darts_final.yaml

⁷https://github.com/walkerning/aw_nas/blob/master/doc/development.md