RANK-NOSH: Efficient Predictor-Based Architecture Search via Non-Uniform Successive Halving

- 作者：Ruochen Wang，Xiangning Chen，Minhao Cheng et.al.
- 机构：UCLA，DiDi AI Labs
- 会议：ICCV 2021
- 地址：https://openaccess.thecvf.com/content/ICCV2021/papers/Wang_RANK-NOSH_Efficient_Predictor-Based_Architecture_Search_via_Non-Uniform_Successive_Halving_ICCV_2021_paper.pdf
- 代码：暂无

论文主要内容

摘要

Predictor-based方法在NAS上效果显著，但是这些方法受到高计算代价限制（训练predictor所需）。本文方法通过削减架构训练的计算budget来提升搜索效率。本文提出NOn-uniform Successive Halving (NOSH)，这是一个层次调度算法来中断训练中表现差的架构来防止浪费budget。相比SOTA Predictor-based方法在不同搜索空间、数据集上减少budges 到原来1/5.

贡献

1. 给了一种新的解决搜索效率的方法
2. 扩展Successive Halving

研究内容

Motivation

- NAS的高效搜索、supernet的weight shareing带来搜索空间限制与inductive biases
- Predictor-based NAS可以解决这些缺点
    a. 训练、评估在pool中的所有架构
    b. fit a surrogate performance predictor

c. 用predictor来propose新的架构并加到pool中

- 但是已有的Predictor-based方法在第一步仍然计算开销大（**对此解决办法主要关注在开发一个需要更小的training pool的predictor，sample efficiency**）
- 这篇文章就主要关注于怎么通过缩小pool中架构个体的training length（就是减少candidate pool中的training epoch）

方法

介绍

Successive Halving：

- 训练一个pool（随机生成的一些configuration），通过一定的schedule逐渐从pool中扔掉performance不好的。主要用在超参搜索技术上。
- 先前的Successive Halving方法是uniform的。即在任一时刻，pool中的candidates都是训练过相同epochs的（因为只是简单的把poor performance的从pool中丢掉）
- 在这篇文章中对Successive Halving进行扩展：新架构会迭代的加入pool，poor performance会保留（用来构造架构pairs）

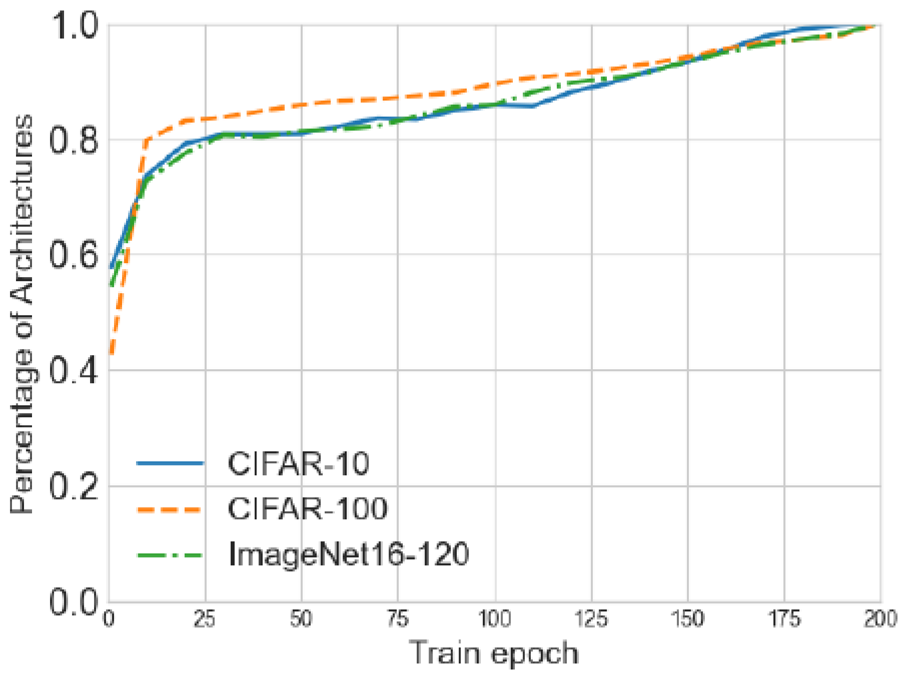

Figure 1: Percentage of architectures with bottom-50% validation accuracy at intermediate epochs that remain at bottom 50% when fully trained on NAS-Bench-201.

说明使用少量epoch训练的结果具有一定的参考价值（训练10epoch，70%的架构满足：如果在开始落后，后面就再也追不上了）

算法流程

NOSH

维护一个金字塔形式的architecture pool（金字塔排序：粗粒度、细粒度的综合，在少train epoch下）

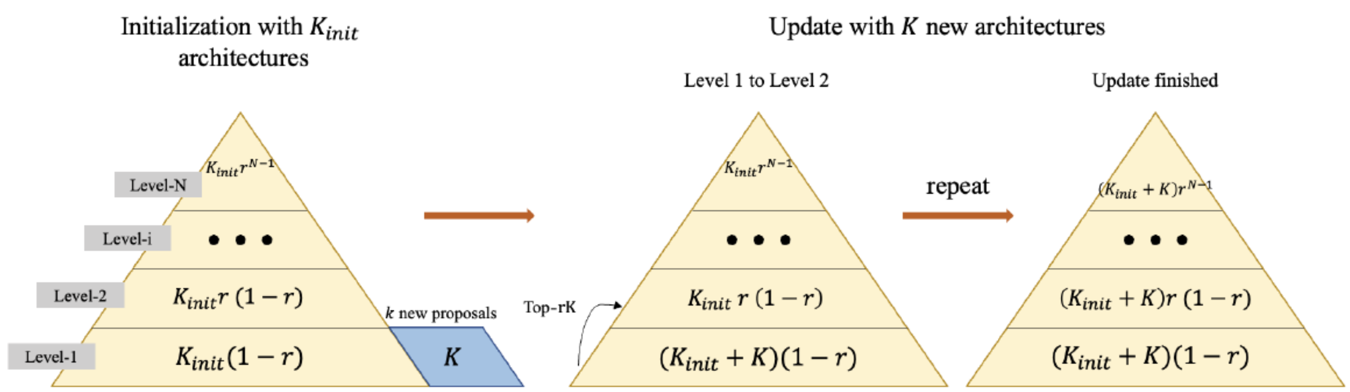

Figure 2: A N-level NOSH pyramid, including its initialization (left) and update (middle & right) processes. Equation inside each level represents the corresponding number of architectures. All architectures in level-i will be trained to epoch $e^{(i)}$. **Left:** During initialization, we populate the pool pyramid. Then we train the predictor and propose K new architectures. **Middle:** We train the K new candidates for $e^{(i)}$ epochs and move Top-rK architectures from level-1 to level-2. **Right:** The pyramid after the update. Then we retrain the predictor and perform the next update, this process continues until a maximum pool size M is achieved.

- 性质：金字塔每一层的architecture的训练epoch一致
- 性质：金字塔level越高（越顶层），训练epoch越多，越充分。金字塔的最顶层是fully triained
- Level i 的architecture被训练次数为 e^i epoch。 $E=\left\{e^{(i)}\right\}_{i=1}^{N}$ ， $e^{(i)}<e^{(i+1)}$ ；move ratio $r\in(0,1)$
- 每次迭代会用当前金字塔candidates pool训练一个predictor，用来propose K 个新architecture，再更新金字塔pool

1. Initialization
    a. K_{init} 个architecture
    b. train这些architecture $e^{(1)}$ 个epochs，并排序
    c. top $K_{init}r$ 的会进一步训练到 $e^{(2)}$ 个epochs且升到 Level-2；bottom $K_{init}(1-r)$ 的留在Level-1.
    d. 重复升level的过程，直到训练epochs达到 $e^{(N)}$ ，即达到金字塔顶层。
    e. 初始化结束后，Level-1会有 $K_{init}(1-r)$ candidates，Level-N会有 $K_{\text{init}}\,r^{(N-1)}$ candidates

2. Update（propose、加入pool）
    a. **训练predictor（ranker-based predictor）**，然后propose K 个new architecture（untrained）
    b. 把这个 K 个架构训练 $e^{(1)}$ 个epochs、加入Level-1。
    c. 排序后，top rK 训练至 $e^{(2)}$ 个epochs、移到 Level-2。
    d. 重复上步直到 Level达到 N
3. 增加Level-0

Table 1: Spearman ranking correlation between architectures ranked by training-free metrics and true validation accuracy on CIFAR-10 in NAS-Bench-201 space.

Prior Scores	Whole Space	Top 1% Architectures
grad_norm [1]	0.58	0.42
jacob_cov [25]	0.73	0.13
mag [35]	0.76	0.37

- 这些代理metric（training-free）用于整个NAS空间，performance rank效果好，但是对top configurations的rank不准确
- 基于这个观察结果，把这个metric用来作为Level-0的rank。top configurations放到更高Level上去refine

区别于一般的successive halving：

- non-uniform：pool中的架构有着不同训练程度（Level）
- 先前训练中断的架构有机会resume（如果performance能比新propose加入的好）

Algorithm 1: NOSH: Non-Uniform Successive Halving

Input: Candidate pool $\mathcal{S}$, schedule $E = \{e^{(l)}\}_{l=1}^{N}$, move ratio r, Proposal size K (use K_{init} during the initialization round)
Result: updated training pool $\mathcal{S}$
for *level* $l = 0 \sim (N-1)$ **do**
  if $l == 0$ **then**
    Sort all architectures in level-l according to their prior scores;
  else
    Sort all architectures in level-l according to their current validation accuracy;
  end
  Train top rK architectures in level-l to epoch $e^{(l+1)}$ and upgrade them to level-$(l+1)$;
  $K *= r$;
end

Predictor（learning to rank）

Pairwise Ranking

label定义：

$$y(\alpha_1, \alpha_2) = \begin{cases} 1\{e_{\alpha_1} < e_{\alpha_2}\} & e_{\alpha_1} \neq e_{\alpha_2} \\ 1\{acc_{\alpha_1} < acc_{\alpha_2}\} & e_{\alpha_1} = e_{\alpha_2} \end{cases}$$

- 同一Level（同样的epoch训练）的架构：比acc（validation accuracy）大小
- 不同Level的架构直接用Level大小比较

目标：

$$\min_{\mathcal{M}} E_{(\alpha_1,\alpha_2)\sim\mathcal{X}}\left[\ell\left(\mathcal{M}(\alpha_1, \alpha_2), y(\alpha_1, \alpha_2)\right)\right]$$
$$\mathcal{X} = \{(\alpha_1, \alpha_2) \mid \alpha_1 \in \mathcal{S}, \alpha_2 \in \mathcal{S}, \alpha_1 \neq \alpha_2\}$$

- $\mathcal{M}$ 表示rank model， ℓ 表示loss function（BCE）， $\mathcal{S}$ 表示candidates pool

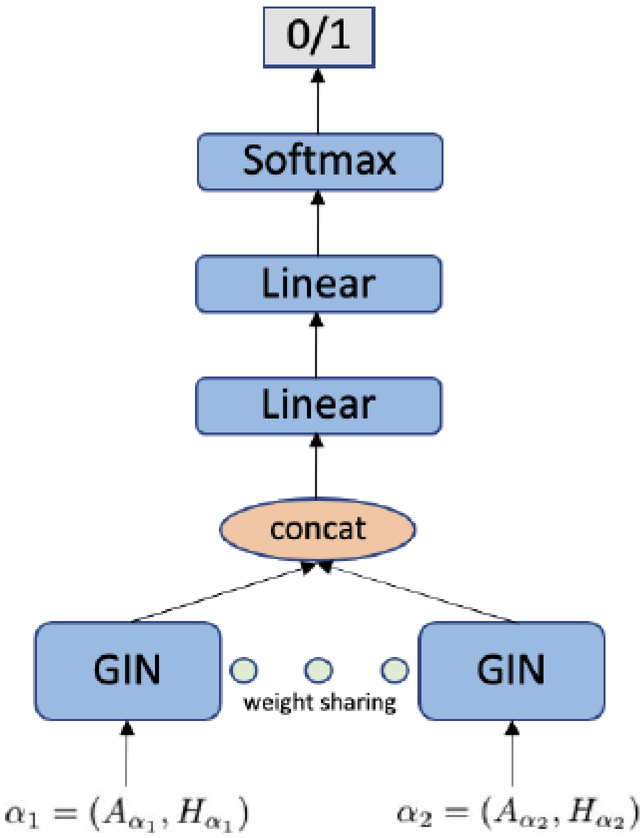

Figure 3: Ranker Network

Propose（Search Algorithm）

- K_{init} 通过从搜索空间里随机采样出来
- K 的选择，由于直接枚举整个search space是不可能的，就是用搜索空间随机选择出来一个子集的枚举rank。加入explicit exploration
  - global ranking选择top $\frac{K}{2}$
  - 剩下的从top $2K$ 中随机选择（排除top $\frac{K}{2}$ 已选择的，保证不重复）

Algorithm 2: RANK-NOSH Main Search

Input: Max candidate pool size M, init pool size K_{init}, proposal size K, schedule $E = \{e^{(l)}\}_{l=1}^{N}$, move ratio r

Result: Discovered best architecture α^*

Randomly select K_{init} architectures and add them to $\mathcal{S}$;

Initialize Pyramid: $\mathcal{S}$ = NOSH($\mathcal{S}$, E, r, K_{init});

$M \mathrel{+}= K_{init}$;

while $|\mathcal{S}| < M$ **do**

  Generate pairwise labels according to Eq. (1);

  Fit the ranker model with labeled $\mathcal{S}$;

  Use the ranker to propose top $min(K, M - |\mathcal{S}|)$ architectures and add them to $\mathcal{S}$;

  Update Pyramid: $\mathcal{S}$ = NOSH($\mathcal{S}$, E, r, K);

S $\boxed{M} \mathrel{+}= K$;

end

$\alpha^* = \arg\max_{\alpha \in \mathcal{S}} Valid\_Acc_{\alpha}$

实验结果

实验设置

- Ranker：用arch2vec **pretrain** GIN encoder，来提高架构的representation
- $r = \frac{1}{2}$ ，Level-0用1/3（no cost下能容纳更多架构）
- metric（prior score）选用magnitude of weights
- 对每个搜索空间（nasbench、DARTS）利用标准full training epoch来确定 schedule E
- K_{init} 为16*3， K 为10*3。(有2/3的架构在Level-0无训练代价)

Results

NAS-Bench-201

E = (1, 2, 3, 12) for CIFAR-10, E =(10, 50, 100, 200) CIFAR-100 and ImageNet16-120 to match the maximum training epochs

Table 2: Comparison with state-of-the-art NAS methods on NAS-Bench-201.

Method	CIFAR-10			CIFAR-100			ImageNet16-120		
	validation	test	budget	validation	test	budget	validation	test	budget
DARTS [23]	39.77 ± 0.00	54.30 ± 0.00	-	38.57 ± 0.00	38.97 ± 0.00	-	18.87 ± 0.00	18.41 ± 0.00	-
SNAS [43]	90.10 ± 1.04	92.77 ± 0.83	-	69.69 ± 2.39	69.34 ± 1.98	-	42.84 ± 1.79	43.16 ± 2.64	-
GDAS [10]	90.01 ± 0.46	93.23 ± 0.23	-	71.14 ± 0.27	70.61 ± 0.26	-	41.70 ± 1.26	41.84 ± 0.90	-
PC-DARTS [44]	89.96 ± 0.15	93.41 ± 0.30	-	67.12 ± 0.39	67.48 ± 0.89	-	40.83 ± 0.08	41.31 ± 0.22	-
ENAS [29]	39.77 ± 0.00	54.30 ± 0.00	-	15.03 ± 0.00	15.61 ± 0.00	-	16.43 ± 0.00	16.32 ± 0.00	-
Prior Score: jacob_cov [25]	89.69 ± 0.73	92.96 ± 0.80	-	69.87 ± 1.22	70.03 ± 1.16	-	43.99 ± 2.05	44.43 ± 2.07	-
Prior Score: mag [35]	89.94 ± 0.34	93.35 ± 0.04	-	70.18 ± 0.66	70.47 ± 0.18	-	42.57 ± 2.14	43.17 ± 2.57	-
RE [30] *	91.04 ± 0.51	93.81 ± 0.46	$1,200$	72.18 ± 0.91	72.06 ± 0.97	$20,000$	45.78 ± 0.72	45.67 ± 0.83	$20,000$
RS [3] *	90.91 ± 0.41	93.69 ± 0.42	$1,200$	71.36 ± 0.84	71.32 ± 0.95	$20,000$	45.26 ± 0.67	45.24 ± 0.84	$20,000$
REINFORCE [41] *	90.32 ± 0.85	93.21 ± 0.76	$1,200$	70.95 ± 1.22	70.87 ± 1.23	$20,000$	44.66 ± 1.44	44.63 ± 1.52	$20,000$
arch2vec-BO [45] *	91.4 ± 0.35	94.24 ± 0.21	$1,200$	73.29 ± 0.41	73.41 ± 0.22	$20,000$	46.27 ± 0.39	46.32 ± 0.27	$20,000$
RANK-NOSH	$\mathbf{91.4 \pm 0.18}$	$\mathbf{94.26 \pm 0.17}$	**292**	$\mathbf{73.49 \pm 0.00}$	$\mathbf{73.51 \pm 0.00}$	$\mathbf{5,550}$	$\mathbf{46.37 \pm 0.0}$	$\mathbf{46.34 \pm 0.0}$	$\mathbf{5,550}$
oracle	91.61	94.37	-	73.49	73.51	-	46.77	47.31	-

* Reproduced by directly searching on every dataset with a candidate pool size of 100 architectures following [45]. Note that the original arch2vec paper [45] measures the search budget in seconds, which translates to approximately 100 architectures on all three datasets.

NAS-Bench-101

Table 4: Comparison with SOTA methods on NAS-Bench-101. We report the avg test accuracy for our method over 10 random seeds.

Methods	Search Budget (#epochs)	Test Accuracy (%)
Prior Score: jacob_conv [25]	-	89.11
Prior Score: mag [35]	-	92.66
Random Search [46]	108,000	93.54
REINFORCE [46]	108,000	93.58
Regularized Evolution [46]	108,000	93.72
NAO [24]	108,000	93.74
BANANAS [40]	54,000	94.08
arch2vec-BO [45]	43,200	94.05
RANK-NOSH	8,400	93.97

本文的方法和 SOTA 方法performance相当，但只有 19% 的 budget.

DARTS Space

总共 10^9 可能的架构，从中随机采样600k架构，在这个子集上实验

CIFAR-10

Search budgets：990 epochs(1.65x DARTS)

Table 3: Comparison with state-of-the-art NAS methods on DARTS Space.

Architecture	Test Error(%) Best	Test Error(%) Avg	Param (M)	Search Budget (#epochs)	Search Method
RSWS [20]	2.71	2.85 ± 0.08	4.3	-	Weight Sharing
DARTS [23]	$2.76 \pm 0.09^{\star}$	-	3.6	-	Weight Sharing
SNAS [43]	-	2.85 ± 0.02	2.8	-	Weight Sharing
BayesNAS [49]	$2.81 \pm 0.04^{\star}$	-	3.4	-	Weight Sharing
ProxylessNAS [4]	$\mathbf{2.08}^{\dagger}$	-	4.0	-	Weight Sharing
ENAS [29]	$2.89^{\dagger}$	-	4.6	-	Weight Sharing
P-DARTS [8]	2.50	-	3.4	-	Weight Sharing
PC-DARTS [44]	$2.57 \pm 0.07^{\star}$	-	3.6	-	Weight Sharing
SDARTS-ADV [6]	-	2.61 ± 0.02	3.3	-	Weight Sharing
Random Search [23]	$3.29 \pm 0.15^{\star}$	-	3.2	2,400	Random
GATES [27]	$2.58^{\dagger}$	-	4.1	64,000	Predictor
BRP-NAS (high) [12]	-	2.59 ± 0.11	-	36,000	Predictor
BRP-NAS (med) [12]	-	2.66 ± 0.09	-	18,000	Predictor
BANANAS [40]	2.57	2.64	3.6	5,000	Predictor
arch2vec-BO [45]	**2.48**	2.56 ± 0.05	3.6	5,000	Predictor
RANK-NOSH	2.50	$\mathbf{2.53 \pm 0.02}$	3.5	**990**	Predictor

† Obtained on different search spaces than DARTS.

* Error bars are computed by retraining the best discovered architecture multiple times.

ImageNet

用搜出来的架构放在ImageNet上评估（用transfer learning setting）

Table 5: Transfer learning results on ImageNet

Architecture	Test Error(%)	Params (M)
NASNet-A [51] *	26.0	5.3
AmoebaNet-A [31] *	25.5	5.1
PNAS [22] *	25.8	5.1
SNAS [43] *	27.3	4.3
DARTS [23] *	26.7	4.7
SDARTS-ADV [6]	25.2	4.8
arch2vec-BO [45] *	25.5	5.2
RANK-NOSH	25.2	5.3

* Results obtained from the arch2vec paper [45].

消融实验

实验使用NAS-Bench-201

Train-free Prior scores

验证是否直接用score就能完成NAS任务。

- 直接从搜索空间中采样1000个架构，并用prior score最高来选取best架构
- 结果明显poor performance（比random search差）
- 原因是这些 scores 不能够区分top architecture

Comparison with Early Stopping

验证是否直接使用简单的 Early Stopping 就行（任何一个架构都不会被完全训练）

Table 6: Validation accuracy (%) of the final architectures obtained by RANK-NOSH v.s. arch2vec-BO with early stopping on NAS-Bench-201.

Dataset	**Search Budget**	**arch2vec-BO**	**RANK-NOSH**
CIFAR-10	5,550	91.00 ± 0.61	91.60 ± 0.02
	2,969	90.35 ± 0.62	91.56 ± 0.07
CIFAR-100	5,550	73.23 ± 0.61	73.49 ± 0.00
	2,969	71.88 ± 1.19	73.44 ± 0.09
ImageNet16-120	5,550	46.08 ± 0.75	46.37 ± 0.00
	2,969	45.10 ± 1.07	46.43 ± 0.21

结论：

- Early stop的budget越小，结果越差，variance也越大
- 简单使用early-stopping并没有达到文章的性能
- RANK-NOSH方法具有更小的 variance

NOSH Schedules

定义资源分配的超参有两个 E 和 r

Table 7: Validation Accuracy of final architectures from RANK-NOSH on CIFAR-10 under various schedules and move ratios. Our method is relatively stable across various E and r.

E	**Search Budget**	**Valid Accuracy (%)**
(10,50,200)	6,750	91.60 ± 0.03
(10,50,100,200)	5,550	91.60 ± 0.02
(5,25,50,200)	4,075	91.59 ± 0.03
(5,10,25,200)	3,400	91.57 ± 0.06

(a) Under different E

r	**Search Budget**	**Valid Accuracy (%)**
0.7	9,750	91.58 ± 0.06
0.6	7,400	91.59 ± 0.06
0.5	5,550	91.60 ± 0.02
0.4	4,100	91.58 ± 0.08
0.3	2,950	91.40 ± 0.16

(b) Under different r

- 固定 $r = \frac{1}{2}$ ，改变 E
  - 结论：stable（无视 Levels数量与 epoch 间隔）
- 固定 E 为(10, 50, 100, 200)，改变 r
  - 结论：robust
- 建议：选择 $r = \frac{1}{2}$ ， E 根据不同搜索budgets来确定

> Moreover, the proposed framework could be extended to other applications. For instance, RANK-NOSH can be applied to **hyperparameter optimization** by **concatenating the hyperpa-rameters with the architecture embeddings**