Sample-Efficient Neural Architecture Search by Learning Action Space

Linnan Wang[1] **Saining Xie**[2] **Teng Li**[2] **Rodrigo Fonseca**[1] **Yuandong Tian**[2]
[1]Brown University [2]Facebook AI Research

Abstract

Neural Architecture Search (NAS) has emerged as a promising technique for automatic neural network design. However, existing NAS approaches often utilize manually designed action space, which is not directly related to the performance metric to be optimized (e.g., accuracy). As a result, using manually designed action space to perform NAS often leads to sample-inefficient explorations of architectures and thus can be sub-optimal. In order to improve sample efficiency, this paper proposes Latent Action Neural Architecture Search (LaNAS) that learns the action space to recursively partition the architecture search space into regions, each with concentrated performance metrics (*i.e.*, low variance). During the search phase, as different architecture search action sequences lead to regions of different performance, the search efficiency can be significantly improved by biasing towards the regions with good performance. On the largest NAS dataset NasBench-101, our experimental results demonstrated that LaNAS is $22\times$, $14.6\times$ and $12.4\times$ more sample-efficient than random search, regularized evolution, and Monte Carlo Tree Search (MCTS) respectively. When applied to the open domain, LaNAS finds an architecture that achieves SoTA 98.0% accuracy on CIFAR-10 and 75.0% top1 accuracy on ImageNet (mobile setting), after exploring only 6,000 architectures.

1 Introduction

During the past two years, there has been a growing interest in Neural Architecture Search (NAS) that aims to automate the laborious process of designing neural networks. Architectures found by NAS have achieved remarkable results in image classification [1, 2], object detection and segmentation [3, 4, 5], as well as other domains such as language tasks [6, 7].

Starting from hand-designed discrete model space and action space, NAS utilizes search techniques to explore the search space and find the best performing architectures with respect to a single or multiple objectives (*e.g.*, accuracy, latency, or memory), and preferably with minimal search cost. However, one common issue faced by the previous works on NAS is that the action space needs to be manually designed. The action space proposed by Zoph *et al.* [8] involves sequential actions to construct a network, such as selecting two nodes, and choosing their operations. Other prior works including gradient-based [9, 10], reinforcement learning based [1, 11], evolution-based [2, 12], and MCTS-based [13, 14] approaches, all use manually designed action spaces. As suggested in [15, 16, 17], action space design alone can be critical to network performance.

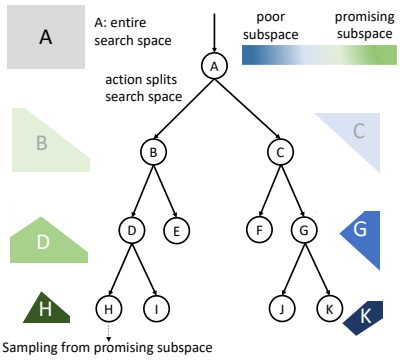
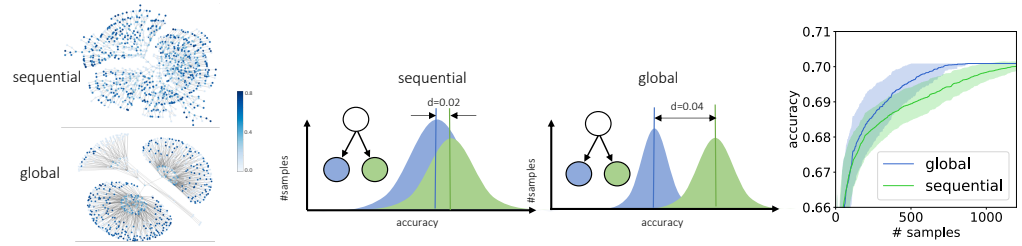


Figure 1: Latent Action Neural Architecture Search. Starting from the entire model space, at each search stage we learn an action (or a set of *linear constraints*) to separate good from bad models, providing distinctive rewards for better searching.

Preprint. Under review.



(a) Visualization of MCTS search trees (b) Reward distribution at two child nodes (c) Search efficiency

Figure 2: **Illustration of motivation**: (a) visualizes the MCTS search trees using `sequential` and `global` action space. The node value (*i.e.* accuracy) is higher if the color is darker. (b) For a given node, the reward distributions for its children. d is the average distance over all nodes. `global` better separates the search space by network quality, and provides distinctive reward in recognizing a promising path. (c) As a result, `global` finds the best network much faster than `sequential`.

Furthermore, it is often the case that manually designed action space is not related to the performance that needs to be optimized. In Sec 2, we demonstrate an example where subtly different action space can lead to significantly different search efficiency. Finally, unlike games that generally have a predefined action space (e.g., Atari, Chess and Go), in NAS, it is the final network that matters rather than the specific path of search, which gives a large playground for action space learning.

Based on above observations, we propose Latent Action Neural Architecture Search(LaNAS) that learns the action space to maximize search efficiency for a given performance metric. While previous methods typically construct an architecture from an empty network by sequentially applying predefined actions, LaNAS takes a dual approach and treats each action as a *linear constraint* which intersects with the current model space to yield a smaller region. Our goal is to find a high-performance sub-region, once multiple actions are applied to the entire model space. To achieve this goal, LaNAS iterates between *learning* and *exploration* stage. In the learning stage, each action in LaNAS is learned to partition the model space into high-performance and low-performance regions, to achieve accurate performance prediction. In exploration stage, LaNAS applies MCTS on the learned action space to get more model architectures and the corresponding performance. The learned action space provides an informed guide for the search algorithm, while the exploration in MCTS collects more data to progressively bias the learned space towards more performance-promising regions. The iterative process is jump-started by first collecting a few random models.

We show that LaNAS yields a tremendous acceleration on a diverse set of benchmark tasks, including publicly available NASBench-101 (420,000 NASNet models trained on CIFAR-10)[18], our self-curated ConvNet-60K (60,000 plain VGG-style ConvNets trained on CIFAR-10), and LSTM-10K (10,000 LSTM cells trained on PTB). Our algorithm consistently finds the best performing architecture on all three tasks with an average of at least an order of fewer samples than random search, vanilla MCTS, and Regularized Evolution. In the open domain search scenario, our algorithm finds a network that achieves 98.0% accuracy on CIFAR-10 and 75.0% top1 accuracy (mobile setting) on ImageNet in only 6,000 samples, using 4.4x fewer samples and achieving higher accuracy than AmoebaNet [2]. Moreover, we empirically demonstrate that the learned latent actions can transfer to a new search task to further boost efficiency. Finally, we provide empirical observations to illustrate the search dynamics and analyze the behavior of our approach. We also conduct various ablation studies in together with a partition analysis to provide guidance in determining search hyper-parameters and deploying LaNAS framework in practice.

2 A Motivating Example

To demonstrate the importance of action space in NAS, we start with a motivating example. Consider a simple scenario of designing a plain Convolutional Neural Network (CNN) for CIFAR-10 image classification. The primitive operation is a Conv-ReLU layer. Free structural parameters that can vary include network depth $L = \{1, 2, 3, 4, 5\}$, number of filter channels $C = \{32, 64\}$ and kernel size $K = \{3 \times 3, 5 \times 5\}$. This configuration results in a search space of 1,364 networks. To perform the search, there are two natural choices of the action space: `sequential` and `global`. `sequential` comprises actions in the following order: adding a layer l, setting kernel size K_l, setting filter channel

C_l. The actions are repeated L times. On the other hand, `global` uses the following actions instead: {Setting network depth L, setting kernel size $K_{1,\ldots,L}$, setting filter channel $C_{1,\ldots,L}$}. For these two action spaces, MCTS is employed to perform the search. Note that both action spaces can cover the entire search space but have very different search trajectories.

Fig. 2(a) visualizes the search for these two action spaces. Actions in `global` clearly separates desired and undesired network clusters, while actions in `sequential` lead to network clusters with a mixture of good or bad networks in terms of performance. As a result, the overall distribution of accuracy along the search path (Fig. 2(b)) shows concentration behavior for `global`, which is not the case for `sequential`. We also demonstrate the overall search performance in Fig.2(c). As shown in the figure, `global` finds desired networks much faster than `sequential`.

This observation suggests that changing the action space can lead to very different search behavior and thus potentially better sample efficiency. In other words, an early exploration on the network depth is critical. Increasing the depth is an optimization direction that can potentially lead to better model accuracy. One might come across a natural question from this motivating example. Is it possible to find a principle way to distinguish a good action space from a bad action space in NAS? Is it possible to *learn an action space* such that it can best fit the performance metric to be optimized?

3 Learning Latent Action Space

In this section, we present LaNAS, which comprises two phrases: (1) search phase (2) learning phase. LaNAS iteratively learns an action space and explores with the current action space. In Fig. 3, we illustrate a high level description of LaNAS, of which the corresponding algorithms are further described in Alg.1.

3.1 Learning Phrase

In the learning phrase at iteration t, we have a fixed dataset $D_t = \{(\mathbf{a}_i, v_i)\}$ obtained from previous explorations. Each data point $(\mathbf{a}_i, v_i)$ in D_t has two components: $\mathbf{a}_i$ represents network attributes (e.g., depth, number of filters, kernel size, connectivity, etc) and v_i represents the performance metric estimated from training (or from pre-trained dataset like NASBench-101). Our goal is to learn a good action space from the dataset to guide future exploration as well as to find the model with the desired performance metric efficiently.

Starting from the entire model/architecture space Ω, we recursively (and greedily) split it into smaller regions such that the estimation of performance metric becomes more accurate. This helps us prune away poor regions as soon as possible and increase the sample efficiency of architecture search.

In particular, we model the recursive splitting process as a tree. The root node corresponds to the entire model space Ω, while each tree node j corresponds to a sub-region Ω_j (Fig. 1). At each tree node j, we partition Ω_j into disjoint regions $\Omega_j = \cup_{k \in \mathrm{ch}(j)} \Omega_k$, such that on each child region Ω_k, the estimation of performance metric $V(\Omega_k)$ is the most accurate (or equivalently, has lowest variance).

At each node j, we learn a classifier to split the model space. The linear classifier takes the portion of the dataset that falls into its own region $D \cap \Omega_j$, and output m_j different outcomes, each corresponding to one possible action at the current node j. To minimize the variance of $V(\Omega_k)$ for all child nodes, we learn a linear regressor f_j that minimizes $\sum_{i \in D \cap \Omega_j} (f_j(a_i) - v_i)^2$. Once learned, we use f_j to predict an estimated metric $\hat{v}_i = f_j(a_i)$ for attributes $\mathbf{a}_i$, sort them and partition them into m_j parts. For convenience, we always send network attributes with the best performance to the leftmost child, and so on. The partition thresholds, combined with f_j, become the classifier at node j.

Note that we cannot use v_i directly for partition, since during the search phrase, new architecture $\mathbf{a}_i$ won't have v_i available until it has been trained and evaluated. Instead, we use $\hat{v}_i$ to decide which child node $\mathbf{a}_i$ falls into, and explore branches of the subtree that is likely to give higher performance.

3.2 Search Phrase

Once action space is learned, the search phase follows. It uses the learned action space to explore more architectures as well as their performance. Note that in the learning phrase, we use a fixed (and

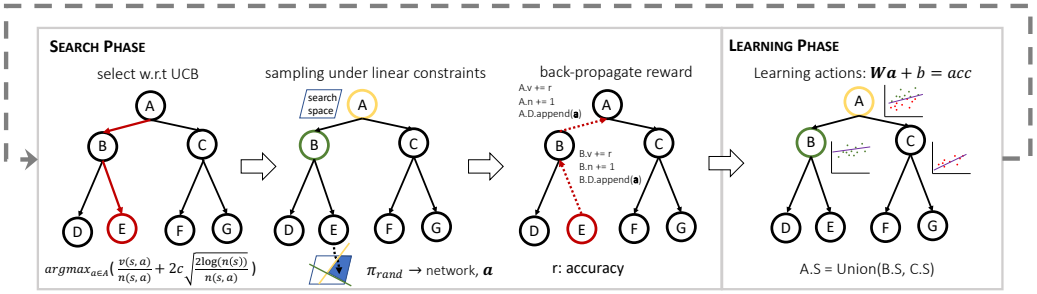


Figure 3: **An overview of LaNAS**: LaNAS is an iterative algorithm in which each iteration comprises a search phase and learning phase. The search phase uses MCTS to samples networks, while the learning phase learns a linear model between network hyper-parameters and their accuracies.

static) tree structure, and what we learn is the different decisions at each node. Therefore, during the search phrase we need to decide which tree branches to try first.

Following the construction of the classifier at each node, a trivial search strategy can be used to evaluate $\mathbf{a}_i$ at each node j, and send it to different child node according to the thresholds. However, this is not a good strategy since it only *exploits* the current action space, which is learned from the current dataset and may not be optimal. There can be good model regions that are hidden in the right (or bad) leaves that need to be explored.

In order to overcome this issue, we use Monte Carlo Tree Search (MCTS) as the search method, which has the characteristics of adaptive exploration, and has shown superior efficiency in various tasks. MCTS keeps track of visiting statistics at each node to achieve the balance between exploitation of existing good children and exploration of new children. In lieu of MCTS, our search phase also has *select*, *sampling* and *backpropagate* stages. LaNAS skips the *expansion* stage in regular MCTS since the connectivity of our search tree is static. Note that we reset all the visitation counts when a new search phase starts, since the counts from the last search phase corresponds to the old action space. In the first iteration, when there is no learned action space, we random sample the model space to get jump started.

3.3 LaNAS Procedures

Algorithm 1: LaNAS search procedure.

Data: specifications of the search domain
Function *get_ucb* ($\bar{X}_{next}$, n_{next}, n_{curt})
  $c = 0.5$
  return $\frac{\bar{X}_{next}}{n_{next}} + 2c\sqrt{\frac{2log(n_{curt})}{n_{next}}}$ **if** $n_{next} \neq 0$ **else** $+\infty$
begin
  while *acc* < *target* **do**
    for $i = 1 \rightarrow \#selects$ **do**
      $leaf, path = ucb\_select(root)$
      $constraints =$ $get\_constraints(path)$
      $network =$ $sampling(constraints)$
      $acc = network.train()$
      $back - propagate(network, acc)$
    for $n \in Tree.N$ **do**
      $n.g.train()$

Algorithm 2: Subroutines in Alg. 1.

Function *get_constraints* (*s_path*)
  constraints = []
  for $node \in s\_path$ **do**
    $\mathbf{W}, b = node.g.params()$, $\bar{X} = node.\bar{X}$
    if *node* on left **then**
      $constraints.add(\mathbf{Wa} + b \geq \bar{X})$
    else
      $constraints.add(\mathbf{Wa} + b < \bar{X})$
  return *constraints*
Function *ucb_select* ($c = root$)
  $path = []$
  while *c not leaf* **do**
    $path.add(c)$
    $l_{ucb} =$ $get\_ucb(c.left.\bar{X}, c.left.n, c.n)$, $r_{ucb} =$ $get\_ucb(c.right.\bar{X}, c.right.n, c.n)$
    if $l_{ucb} > r_{ucb}$ **then** $c = c.left$ **else** $c = c.right$
  return $path, c$

Search phase: 1) ***select w.r.t UCB***: the calculation of UCB follows the standard definition in [19]. As illustrated in Alg. 1 as the search procedure; the input is the number of visits of current and next state, and the next state value $\bar{v}$. The selecting policy π_{ucb} takes the node with the largest UCB score.

Starting from $root$, we follow π_{ucb} to traverse down to a leaf (Alg. 2 line 7-13). 2) ***sampling from a leaf***: in Fig. 3, the sequence of actions impose several linear constraints on the original model space Ω, yield a polytope region for the leaf. Alg. 2 line 1-6 explains how to obtain constraints from an action sequence. There are various techniques [20, 21] to perform uniform sampling in a polytope. Here we use a variant of Markov Chain Monte Carlo (MCMC) sampler to get uniformly distributed samples. 3) ***back-propagate reward***: after training the sampled network, LaNAS back-propagates the reward, i.e. accuracy, to update the node statistics, e.g. n_j, $\bar{v}_j$. It also back-propagates the sampled network so that every parent node keeps the network in S for training.

Learning phase: the learning phase is to train f_j at every node j. With more samples, f becomes more accurate, and $\bar{v}_j$ becomes closer to $\bar{v}_j^*$. However, as MCTS biases towards selecting good samples, LaNAS will zoom into the promising hyper-space, $\bar{v}_j$ goes up, resulting in $\bar{v}_j > \bar{v}_j^*$.

3.4 Partition Analysis

The sample efficiency, i.e. the number of samples to find the global optimal, is closely related to the quality of partition from tree nodes. Here we seek an upper bound for the number of samples in the leftmost node (the most promising sub-domain) to estimate the sampling efficiency.

Assumption 1 *Given a search domain $\mathcal{X}$ having finite samples N, there exists a probabilistic density $P(a < X < b) = \int_a^b f(X)$, where X is the network accuracy.*

Therefore, $N * P(a < X < b)$ gives the number of networks having accuracies in $[a, b]$. Since the accuracy distribution has finite variance, the following holds [22]

$$|E(\overline{X} - M_X)| < \sigma_X \tag{1}$$

$\overline{X}$ is the mean accuracy over $\mathcal{X}$, and M_X is the median accuracy. Note $X \in [0, 1]$, and let's denote $\epsilon = |\hat{X} - \overline{X}|$. Therefore, the maximal distance from $\hat{X}$ to M_X is $\epsilon + \sigma_X$; and the number of networks falling between $\hat{X}$ and M_X is $N * max(\int_{\hat{X}-\epsilon-\sigma_X}^{M_X} f(X)dX, \int_{M_X}^{\hat{X}+\epsilon+\sigma_X} f(X)dX)$, denoted as δ. Therefore, the root partitions $\mathcal{X}$ into two sets that have $\leq \frac{N}{2} + \delta$ architectures.

Theorem 1 *Given a search tree with height = h, the sub-domain represented by the leftmost leaf contains at most $2 * \delta_{max}(1 - \frac{1}{2^h}) + \frac{N}{2^h}$ architectures, and δ_{max} is the largest partition error from the node on the leftmost path.*

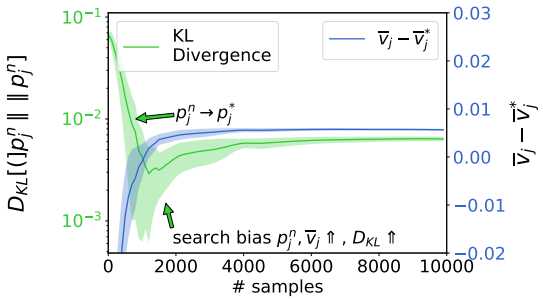
Proof: let's assume the left child is always assigned with the larger partition, and let's recursively apply this all the way down to the leftmost leaf h times, resulting in $\delta^h + \frac{\delta^{h-1}}{2} + \frac{\delta^{h-2}}{2^2} + ... + \frac{N}{2^h} \leq 2 * \delta_{max}(1 - \frac{1}{2^h}) + \frac{N}{2^h}$. δ is related to ϵ and σ_X; note $\delta \downarrow$ with more samples as $\epsilon \downarrow$ as samples $\uparrow$, and σ_X can be estimated from samples. The analysis indicates that LaNAS is approximating a good search region at the speed of $N/2^h$, suggesting 1) the performance improvement will remain near plateau as $h \uparrow$, while the computational costs ($2^h - 1$ nodes) exponentially increase; 2) the performance improvement is limited when N is small. These two points are empirically verified in Fig.6a and Fig.5, respectively.

4 Experiment

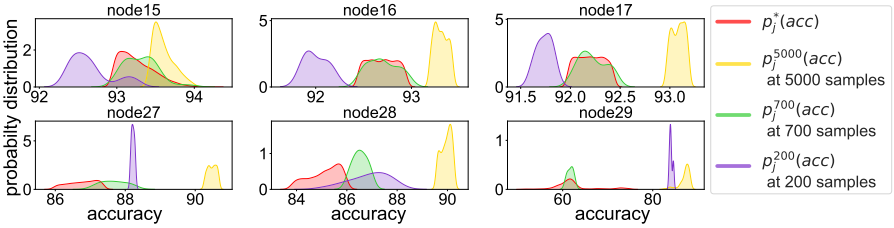
We performed extensive experiments on both offline collected benchmark datasets (e.g., NAS-Bench [18]) and open search domain to validate the effectiveness of LaNAS.

4.1 Analysis of Search Algorithm

We analyze LaNAS using NASBench-101, which contains more than 400K models. The dataset provides us with the true distribution of model accuracy, given any subset of model specifications, or equivalently a collection of actions (or constraints). By construction, left nodes contain regions of the good metric while right nodes contain regions of the poor metric. Therefore, at each node j, we can construct *reference* distribution $p_j^*(v)$ from the entire dataset, by sorting them with respect to the metric, and partition them into different buckets of even size. We compare $p_j^*(v)$ with the empirical



(a) Search dynamics: KL-divergence and mean distance



(b) Search dynamics: sample distribution vs. dataset distribution

Figure 4: **Evaluations of search dynamics**: (a) sample distribution p_j approximates dataset distribution p_j^* when the number of samples $n \in [200, 700]$. The search algorithm then zooms into the promising sub-domain, as shown by the growth of $\bar{v}_j$ when $n \in [700, 5000]$. (b) KL-divergence of p_j and p_j^* dips and bounces back. $\bar{v} - \bar{v}^*$ continues to grow, showing the average metric $\bar{v}$ over different nodes becomes higher when the search progresses.

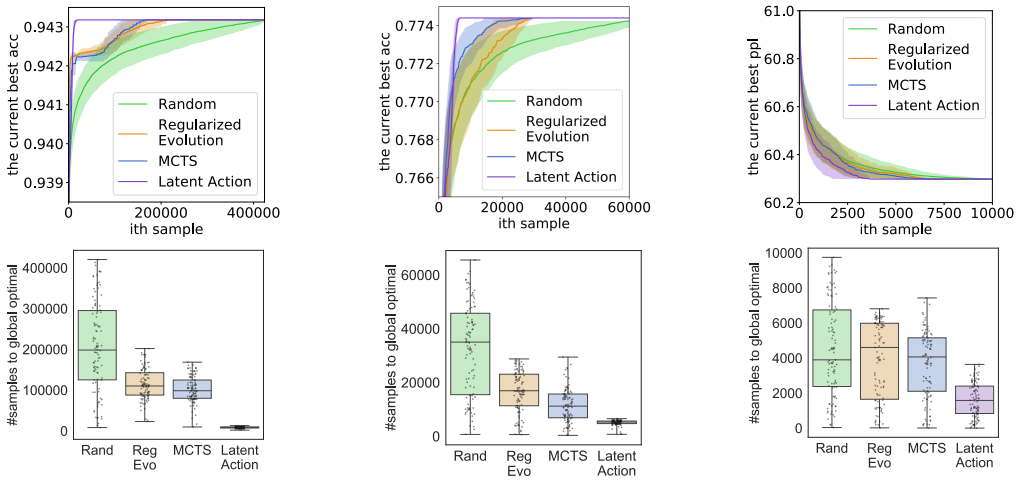
distribution $p_j^n(v)$ estimated from the learned action space, where n is the number of accumulated samples at the node j. To compare $p_j^n(v)$ and $p_j^*(v)$, we use KL-divergence $D_{KL}[p_j^n \| \| p_j^*]$, and their mean value $\bar{v}_j^* = \mathbb{E}_{p_j^*}[v]$ and $\bar{v}_j = \mathbb{E}_{p_j}[v]$.

In our experiments, we use a complete *binary* tree with the height of 5. We label nodes 0-14 as internal nodes, and nodes 15-29 as leaves. By definition, $\bar{v}_{15}^* > \bar{v}_{16}^* ... > \bar{v}_{29}^*$. At the beginning of the search ($n = 200$), p_j^n where j belongs to good sub-domains, e.g. , are expected to be different from p_j^* due to their random initialization. With more samples ($n = 700$), p_j starts to approximate p_j^*, manifested by the increased similarity between p_j^{700} and p_j^*, the transition from p_j^{200} to p_j^{700} in Fig. 4b, and the decreasing D_{KL} in Fig. 4a. This is because MCTS explores the under-explored regions. As the search continues ($n \to 5000$), LaNAS explores deeper into promising sub-domains and p_j^n is biased toward the region with good performance, deviated from even partition which is used to construct p_j^*. As a result, D_{KL} bounces back. These search dynamics demonstrate that our algorithm can adapt to different stages during the course of the search.

4.2 Performance on NAS Datasets

Evaluating on NAS Datasets: We use NASBench-101 [18] as one benchmark that contains over 420K NASNet CNN models. For each network, it records the architecture information and the associated accuracy for fast retrieval by NAS algorithm, avoiding time-consuming model retraining. In addition, we construct two more datasets for benchmarking, ConvNet-60K (plain ConvNet models, VGG-style, no residual connections, trained on CIFAR-10) and LSTM-10K (LSTM cells trained on PTB) to further validate the effectiveness of the proposed LaNAS framework.

Baselines: We compare LaNAS with a few baseline methods that can obtain optimal solution given sufficient explorations. `Random Search` can find the global optimal in expected $n/2$ samples from a dataset of size n, and is dataset-agnostic. `Regularized Evolution` empirically finds the global optimal, and is applied in AmoebaNet [2] that achieves SoTA performance for image recognition. `MCTS` is an anytime search algorithm used in NAS [13] with the global optimality guarantees.



(a) NASBench-420K (b) ConvNet-60K (c) LSTM-10K

Figure 5: **Evaluations of sample-efficiency** on NASBench, ConvNet-60K and LSTM-10K. Each search algorithm is repeated 100 times on each datasets with different random seeds. The top row shows the time-course of the search algorithms (current best accuracy with interquartile range), while the bottom row illustrates the number of samples to reach the global optimal.

Analysis of Results: Fig. 5 demonstrates that LaNAS consistently outperforms the baselines by significant margins on three separate tasks. Particularly, on NASBench. LaNAS is on average using **22x**, **14.6x**, and **12.4x** fewer samples than `Random Search`, `Regularized Evolution`, and `MCTS`, respectively, to find the global optimal. On LSTM, LaNAS still performs the best despite that the dataset is small.

Fig. 4 shows that LaNAS minimizes the variance of reward on a search path, making good networks located on good paths, thereby drastically improving the search efficiency. In contrast, `Random Search` relies on blind search and leads to the worst performance. `Regularized Evolution` utilizes a static exploration strategy that maintains a pool of top 500 architectures for random mutations, which is not guided by previous search experience. `MCTS` builds online models of both performance and visitation counts for adaptive exploration. However, without a good action space, the performance model at each node cannot be highly selective, leading to inefficient search (Fig. 2).

4.3 Performance on Open Domain Search

Table 1: Results on CIFAR-10.

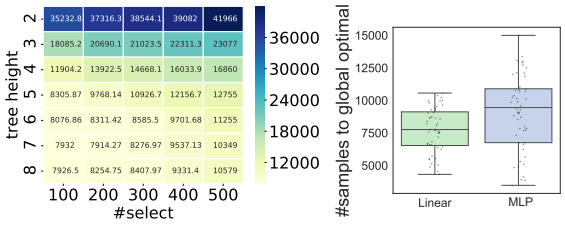
Model	Params	top1 err	M
NASNet-A† [8]	3.3 M	2.65	20000
NASNet-A† [8]	27.6 M	2.40	20000
AmoebaNet-B† [2]	3.2 M	$3.34_{\pm 0.06}$	27000
AmoebaNet-B† [2]	34.9 M	$2.13_{\pm 0.04}$	27000
PNASNet-5 [23]	3.2 M	$3.41_{\pm 0.09}$	1160
NAO [10]	10.6 M	3.18	1000
NAO† [10]	128.0 M	2.11	1000
ENAS† [24]	4.6 M	2.89	N/A
DARTS† [9]	3.3 M	$2.76_{\pm 0.09}$	N/A
LaNet†	3.2 M	$\mathbf{2.53}_{\pm 0.05}$	6000
LaNet†	38.7 M	$\mathbf{1.99}_{\pm 0.02}$	6000

† trained with cutout.
M: number of samples selected.

Table 2: Results on ImageNet (mobile setting)

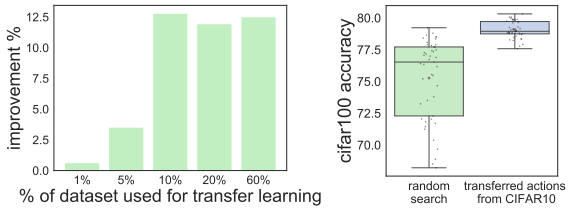
Model	FLOPs	Params	top1 / top5 err
NASNet-A [8]	564M	5.3 M	26.0 / 8.4
NASNet-B [8]	488M	5.3 M	27.2 / 8.7
NASNet-C [8]	558M	4.9 M	27.5 / 9.0
AmoebaNet-A [2]	555M	5.1 M	25.5 / 8.0
AmoebaNet-B [2]	555M	5.3 M	26.0 / 8.5
AmoebaNet-C [2]	570M	6.4 M	**24.3 / 7.6**
PNASNet-5 [23]	588M	5.1 M	25.8 / 8.1
DARTS [9]	574M	4.7 M	26.7 / 8.7
FBNet-C [25]	375M	5.5 M	25.1 / -
RandWire-WS [17]	583M	5.6 M	25.3 / 7.8
LaNet	570M	5.1 M	**25.0 / 7.7**

Table. 1 compares our results in the context of searching NASNet style architecture on CIFAR-10, a common setting used in current NAS research. Experimental setup is further described in Appendix. In only 6000 samples, our best performing architecture (LaNet) demonstrates an average accuracy of 97.47% (#filters = 32, #params = 3.22M) and 98.01% (#filters = 128, #params = 38.7M), which



(a) tree height vs. #select (b) choice of classifier

Figure 6: **Ablation study**: (a) the effect of different tree heights and #select in MCTS. Number in each entry is #samples to reach global optimal. (b) the choice of predictor for splitting search space.



(a) intra-task transfer (b) inter-task transfer

Figure 7: **Latent actions transfer**: learned latent actions can generalize within the same task or across different tasks, to further boost search efficiency.

is better than all existing NAS-based results. It is worth noting that we achieved this accuracy with 4.5x fewer samples than AmoebaNet. Since AmoebaNet and LaNet share the same search space, the saving is purely from our sample-efficient search algorithm. Gradient based methods, and their weight sharing variants, *e.g.* `DARTs` and `NAO`, exhibit weaker performance. We suspect that they are easily trapped into a local optimal, which is observed in [15].

4.4 Transfer Learning

Transfer LaNet to ImageNet: Transferring the best performing architecture (found through searching) from CIFAR10 to ImageNet has already been a standard technique. Following the mobile setting [8], Table. 2 shows that LaNet found on CIFAR-10, when transferred to ImageNet mobile setting (FLOPs are constrained under 600M), achieves competitive performance.

Intra-task latent action transfer: We learn actions from a subset (1%, 5%, 10%, 20% 60%) of NASBench and test their transferability to the remaining dataset, as shown in Fig. 7a. Interestingly, the improvement remain steady after 10%. Consistent with Fig. 4, it is enough to use 10% of the samples to learn the action space.

Inter-task latent action transfer: We compare 100 architectures selected by LaNAS from sec.4.3 (on CIFAR10) with 100 random trials. Networks are trained for 100 epochs and their performances are compared. Fig. 7b indicates that inter-task action transfer is also beneficial.

4.5 Ablation studies

The effect of tree height and #selects: Fig. 6a relates tree height (h) and the number of selects (#selects) to the search performance. In Fig. 6a, each entry represents #samples to achieve optimality on NASBench, averaged over 100 runs. A deeper tree leads to better performance, since the model space is partitioned by more leaves. Similarly, small #select results in more frequent updates of action space, and thus leads to improvement. On the other hand, the number of classifiers increases exponentially as the tree goes deeper, and a small #selects requires frequent action updates. Therefore, both can significantly increase the computation cost.

Choice of classifiers: Fig.6b shows that using a linear classifier performs better than an multi-layer perceptron (MLP) classifier. This indicates that adding complexity to decision boundary of actions may not help with the performance. Conversely, performance can get degraded due to potentially higher difficulties in optimization.

5 Future Work

Recent work on shared model [10, 24] improves the training efficiency by reusing trained components from the similar previously explored architectures, e.g., weight sharing [9, 10, 24]. Our work focuses on sample-efficiency and is complementary to the above techniques.

To encourage reproducibility in NAS research, various of architecture search baselines have been discussed in [15, 16]. We will also open source the proposed LaNAS framework, together with the three NAS benchmark datasets used in our experiments.

References

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

[2] Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V Le. Regularized evolution for image classifier architecture search. *arXiv preprint arXiv:1802.01548*, 2018.

[3] Golnaz Ghiasi, Tsung-Yi Lin, Ruoming Pang, and Quoc V Le. Nas-fpn: Learning scalable feature pyramid architecture for object detection. *arXiv preprint arXiv:1904.07392*, 2019.

[4] Yukang Chen, Tong Yang, Xiangyu Zhang, Gaofeng Meng, Chunhong Pan, and Jian Sun. Detnas: Neural architecture search on object detection. *arXiv preprint arXiv:1903.10979*, 2019.

[5] Chenxi Liu, Liang-Chieh Chen, Florian Schroff, Hartwig Adam, Wei Hua, Alan Yuille, and Li Fei-Fei. Auto-deeplab: Hierarchical neural architecture search for semantic image segmentation. *arXiv preprint arXiv:1901.02985*, 2019.

[6] Minh-Thang Luong, David Dohan, Adams Wei Yu, Quoc V Le, Barret Zoph, and Vijay Vasudevan. Exploring neural architecture search for language tasks. *arXiv preprint arXiv:1901.02985*, 2018.

[7] David R So, Chen Liang, and Quoc V Le. The evolved transformer. *arXiv preprint arXiv:1901.11117*, 2019.

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

[9] Hanxiao Liu, Karen Simonyan, and Yiming Yang. Darts: Differentiable architecture search. *arXiv preprint arXiv:1806.09055*, 2018.

[10] Renqian Luo, Fei Tian, Tao Qin, Enhong Chen, and Tie-Yan Liu. Neural architecture optimization. In *Advances in neural information processing systems*, pages 7816–7827, 2018.

[11] Bowen Baker, Otkrist Gupta, Nikhil Naik, and Ramesh Raskar. Designing neural network architectures using reinforcement learning. *arXiv preprint arXiv:1611.02167*, 2016.

[12] Esteban Real, Sherry Moore, Andrew Selle, Saurabh Saxena, Yutaka Leon Suematsu, Jie Tan, Quoc V Le, and Alexey Kurakin. Large-scale evolution of image classifiers. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 2902–2911. JMLR. org, 2017.

[13] Linnan Wang, Yiyang Zhao, Yuu Jinnai, Yuandong Tian, and Rodrigo Fonseca. Alphax: exploring neural architectures with deep neural networks and monte carlo tree search. *arXiv preprint arXiv:1903.11059*, 2019.

[14] Renato Negrinho and Geoff Gordon. Deeparchitect: Automatically designing and training deep architectures. *arXiv preprint arXiv:1704.08792*, 2017.

[15] Christian Sciuto, Kaicheng Yu, Martin Jaggi, Claudiu Musat, and Mathieu Salzmann. Evaluating the search phase of neural architecture search. *arXiv preprint arXiv:1902.08142*, 2019.

[16] Liam Li and Ameet Talwalkar. Random search and reproducibility for neural architecture search. *arXiv preprint arXiv:1902.07638*, 2019.

[17] Saining Xie, Alexander Kirillov, Ross Girshick, and Kaiming He. Exploring randomly wired neural networks for image recognition. *arXiv preprint arXiv:1904.01569*, 2019.

[18] Chris Ying, Aaron Klein, Esteban Real, Eric Christiansen, Kevin Murphy, and Frank Hutter. Nas-bench-101: Towards reproducible neural architecture search. *arXiv preprint arXiv:1902.09635*, 2019.

[19] Peter Auer, Nicolo Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine learning*, 47(2-3):235–256, 2002.

[20] Edward I George and Robert E McCulloch. Variable selection via gibbs sampling. *Journal of the American Statistical Association*, 88(423):881–889, 1993.

[21] Radford M Neal et al. Slice sampling. *The annals of statistics*, 31(3):705–767, 2003.

[22] Colin Mallows. Letters to the editor. *The American Statistician*, 45(3):256–262, 1991.

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

[24] Hieu Pham, Melody Guan, Barret Zoph, Quoc Le, and Jeff Dean. Efficient neural architecture search via parameter sharing. In *International Conference on Machine Learning*, pages 4092–4101, 2018.

[25] Bichen Wu, Xiaoliang Dai, Peizhao Zhang, Yanghan Wang, Fei Sun, Yiming Wu, Yuandong Tian, Peter Vajda, Yangqing Jia, and Kurt Keutzer. Fbnet: Hardware-aware efficient convnet design via differentiable neural architecture search. *arXiv preprint arXiv:1812.03443*, 2018.