Sample-Efficient Neural Architecture Search by Learning Actions for Monte Carlo Tree Search

Linnan Wang, Saining Xie, Teng Li, Rodrigo Fonseca, Yuandong Tian *Member, IEEE*

Abstract—Neural Architecture Search (NAS) has emerged as a promising technique for automatic neural network design. However, existing MCTS based NAS approaches often utilize manually designed action space, which is not directly related to the performance metric to be optimized (e.g., accuracy), leading to sample-inefficient explorations of architectures. To improve the sample efficiency, this paper proposes Latent Action Neural Architecture Search (LaNAS), which learns actions to recursively partition the search space into good or bad regions that contain networks with similar performance metrics. During the search phase, as different action sequences lead to regions with different performance, the search efficiency can be significantly improved by biasing towards the good regions. On three NAS tasks, empirical results demonstrate that LaNAS is at least an order more sample efficient than baseline methods including evolutionary algorithms, Bayesian optimizations and random search. When applied in practice, both one-shot and regular LaNAS consistently outperforms existing results. Particularly, LaNAS achieves 99.0% accuracy on CIFAR-10 and 80.4% top1 accuracy at 600 MFLOPS on ImageNet in only 800 samples, significantly outperforming AmoebaNet with $33\times$ fewer samples.

Index Terms—one-shot Neural Architecture Search, Monte Carlo Tree Search

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. Starting from discrete state space and action space, NAS utilizes search techniques to explore the search space and find the best performing architectures concerning single or multiple objectives (*e.g.*, accuracy, latency, or memory), and preferably with minimal search cost.

While it is impressive to find a good network architecture in a large search space, one component that is often overlooked is how to design the action space. Most previous methods use manually designed action space [1], [2], [3]. Sec 2 provides a simple example that different action space can significantly improve the search efficiency, by focusing on promising regions (e.g., deep networks rather than shallow ones) at the early stage of search. Furthermore, compared to games that come up with a predefined action space (e.g., Atari or Go), learning action space is more suitable for NAS where the final network matters rather than specific action paths.

Based on the above observations, we propose LaNAS that learns *latent actions* and prioritizes the search accordingly. To achieve this goal, LaNAS iterates between *learning* and *searching* stage. In the learning stage, LaNAS models each action as a *linear constraint* that optimally bi-partitions the search space Ω into high-performing and low-performing regions. Such partitions can be done recursively, yielding a hierarchical tree structure, where some leaf nodes contain very promising regions, e.g. Fig. 1. In the searching stage, LaNAS applies Monte Carlo Tree Search (MCTS) on the tree structure to sample architectures. The learned actions provide an abstraction of search space for MCTS to do an efficient search, while MCTS collects more data with adaptive exploration to progressively refine the learned actions and partitions. The iterative process is jump-started by first collecting a few random samples.

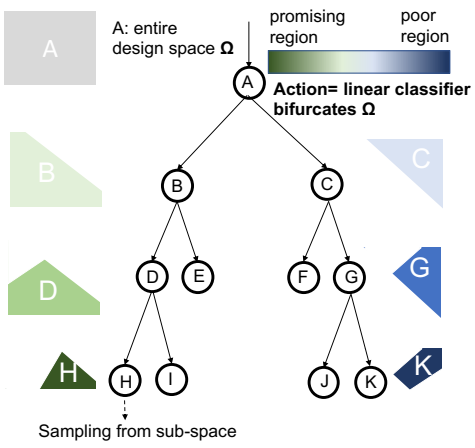
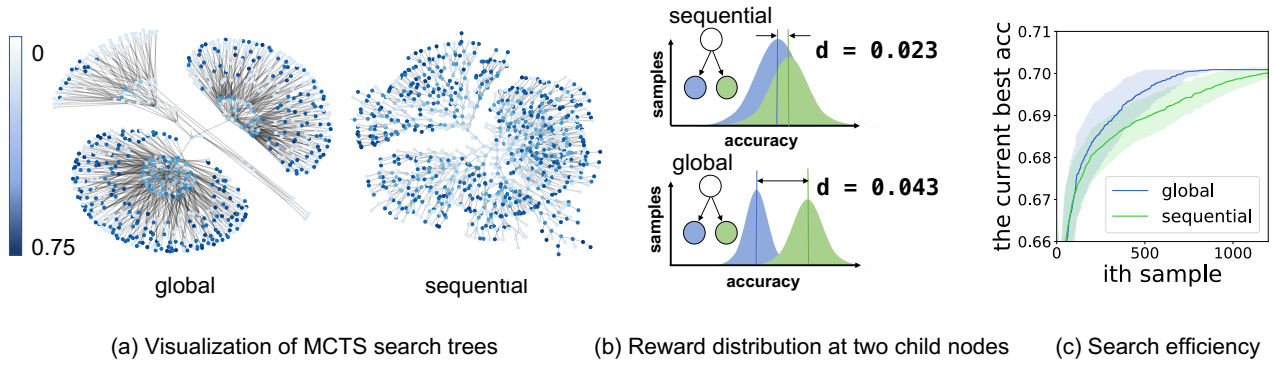


Fig. 1: Starting from the entire model space, at each search stage we learn an action (or a set of *linear constraints*) to separate good from bad models for providing distinctive rewards for better searching. Fig. 10 in appendix provides a visualization of the partitioning process in LaNAS.

Our empirical results show that LaNAS fulfills many desiderata proposed by [4] as a practical solution to NAS: 1) ***Strong final performance***: we show that LaNAS consistently yields the lowest regret on a diverse of NAS tasks using at least an order of fewer samples than baseline methods including MCTS, Bayesian optimizations, evolutionary algorithm, and random search. In practice, LaNAS finds a network that

- *L. Wang and R. Fonseca are with the Department of Computer Science, Brown University, Providence, RI, 02905.*
- *S. Xie, T. Li, Y. Tian are with Facebook AI Research, Menlo Park, CA, 94025.*



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

Fig. 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`. This motivates us to learn actions to partition the search space for the efficient architecture search.

achieves SOTA 99.0% accuracy on CIFAR-10 and 81% top1 accuracy (mobile setting) on ImageNet in only 800 samples, using 33× fewer samples and achieving higher accuracy than AmoebaNet [5]. 2) ***Use of parallel resources***: LaNAS scales to, but not limited to, 500 GPUs in practice. 3) ***Robustness & Flexibility***: The tree height and the exploration factor in UCB are only hyper-parameters in LaNAS, and we also conduct various ablation studies in together with a partition analysis to provide guidance in determining search hyper-parameters and deploying LaNAS in practice.

LaNAS is an instantiation of Sequential Model-Based Optimization (SMBO) [6], a framework that iterates between optimizing an acquisition function to find the next architecture to explore and obtaining the true performance of that proposed architecture to refine the acquisition. Since evaluating a network is expensive, SMBO is an efficient search framework for generalizing architecture performance based on collected samples to inform the search. Table. 1 summarizes the attributes of existing search methods. Compared to Bayesian methods such as TPE, SMAC, and BOHB, LaNAS uses previous search experience to learn latent actions, which converts complicated non-convex optimization of the acquisition functions into a simple traversal of hierarchical partition tree in the next search iterations, while still precisely captures the promising region for the sample proposal, therefore more efficient than Bayesian methods especially in high-dimensional tasks. We elaborate key differences to baseline methods (Table. 1) in Sec. 4.1.4.

TABLE 1: Methods used in experiments and their attributes.

Methods	SMBO	sampling mechanism	scalable to $\lvert\Omega\rvert \sim 10^{20}$	global search
HyperBand [7]	X	successive halving	√	X
BOHB [4]	√	non-convex optimization	X	X
SMAC [8]	√	non-convex optimization	X	√
TPE [9]	√	non-convex optimization	X	√
RE [10]	X	top-k random	√	X
Random [11]	X	random	√	√
MCTS [12]	X	UCB and search tree	√	√
LaNAS	√	UCB and search tree	√	√

SMBO: Sequential Model Based Optimizations
$|\Omega|$ is the size of search space

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 this case, an early exploration of network depth is critical. Increasing depth is an optimization direction that can potentially lead to better model accuracy. One might come across a natural question

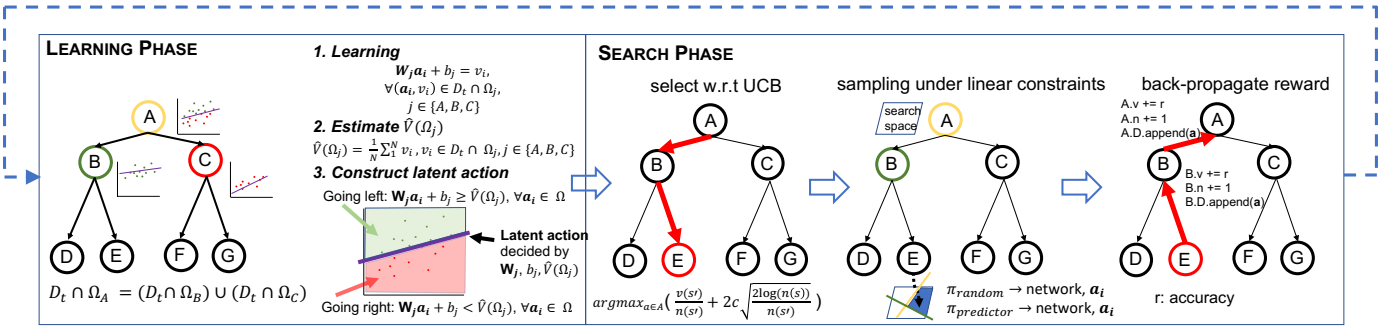


Fig. 3: **An overview of LaNAS**: Each iteration of LaNAS comprises a search and learning phase. The search phase uses MCTS to sample networks, while the learning phase learns a linear model between network hyper-parameters and accuracies.

from this motivating example: is it possible to find a principle way to distinguish a good action space from a bad action space for MCTS? Is it possible to *learn an action space* such that it can best fit the performance metric to be optimized?

3 Learning Latent Actions

In this section, we describe LaNAS that learns the action space for MCTS. Fig. 3 presents a high-level description of LaNAS, of which the corresponding algorithms are further described in Alg.1 in Appendix. The following describes a list of notations used in this paper.

a_i : the ith sampled architecture
v_i : the performance metric of $\mathbf{a}_i$
D_t : the set of collected$(\mathbf{a}_i, v_i)$ at the search step t
Ω : the entire search space
Ω_j : the partition of Ω represented by the tree node j
$D_t \cap \Omega_j$: samples classified in Ω_j
$V(\Omega_j)$: the mean performance metric inΩ_j
$\hat{V}(\Omega_j)$: the estimated $V(\Omega_j)$ from $D_t \cap \Omega_j$
$f_j(\mathbf{a}_i)$: predicted performance by the regressor on node j
$n(s)$: the #visits of tree node s
$v(s)$: the value of tree node s

3.1 Learning Phase

In the learning phase at iteration t, we have a dataset $D_t = \{(\mathbf{a}_i, v_i)\}$ obtained from previous explorations. Each data point $(\mathbf{a}_i, v_i)$ has two components: $\mathbf{a}_i$ represents an architecture in specific encoding (e.g., width=512 and depth=5, etc) and v_i represents the performance metric estimated from training (or from pre-trained dataset such as NASBench-101, or estimated from a supernet in one-shot NAS). Our goal is to learn a good action space from D_t that splits Ω so that architectures with difference performance can be partitioned into high-performing and low-performing partitions. The performance is similar within each partition, but across partitions, the performance of partitions can be easily ranked from low to high. This split can be done recursively to form a hierarchy. Such partitions can help prioritize the search towards more promising regions, and improve the sample efficiency.

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 region Ω_j (Fig. 1). At each tree node j, we partition Ω_j into disjoint regions $\Omega_j = \cup_{k \in \text{ch}(j)} \Omega_k$, 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 regressor that embodies a latent action to split the model space Ω_j. The linear regressor takes the portion of the dataset that falls into its own region $D_t \cap \Omega_j$, then the average performance of a region is estimated by

$$\hat{V}(\Omega_j) = \frac{1}{N} \sum_{v_i \in D_t \cap \Omega_j} v_i \tag{1}$$

To minimize the variance of $V(\Omega_k)$ for all child nodes, we learn a linear regressor f_j

$$\underset{(\mathbf{a}_i, v_i) \in D_t \cap \Omega_j}{\text{minimize}} \sum (f_j(\mathbf{a}_i) - v_i)^2 \tag{2}$$

line 6 → 7 in Alg.1 (appendix) further explains the procedures. With more samples, f_j becomes more accurate, and $\hat{V}(\Omega_j)$ becomes closer to $V(\Omega_j)$.

Once learned, the parameters of f_j and $\hat{V}(\Omega_j)$ form a linear constraint that bifurcates Ω_j into a good region ($> \hat{V}(\Omega_j)$) and a bad region ($\leq \hat{V}(\Omega_j)$). A visualization of this process is available in Fig. 3 (learning phase). For convenience, the left child always represents the good region. The partition threshold $\hat{V}(\Omega_j)$, combined with parameters of f_j, forms two latent actions at node j,

$$\textbf{go-left} : f_j(\mathbf{a}_i) > \hat{V}(\Omega_j)$$
$$\textbf{go-right} : f_j(\mathbf{a}_i) \leq \hat{V}(\Omega_j), \forall \mathbf{a}_i \in \Omega$$

Because the tree recursively splits Ω, partitions represented by leaves follow $V(\Omega_{leftmost}) > ... > V(\Omega_{rightmost})$, with the leftmost leaf representing the most promising partition. Experiments in Sec. 4.3 validate the effectiveness of proposed method in achieving the target behavior.

Note that we need to initialize each node classifier properly with a few random samples to establish an initial boundary in the search space. An ablation study on the number of samples for initialization is provided in Fig. 7(c).

3.2 Search Phase

Once actions are learned, the search phase follows. The search uses learned actions to sample more architectures $\mathbf{a}_i$, and get v_i by training, then store $(\mathbf{a}_i, v_i)$ in dataset D_t to refine the action space in the next learning phase. Note that in the search phase, the tree structure and the parameters of those classifiers are fixed and static. The search phase learns to decide which region Ω_j on tree leaves to sample $\mathbf{a}_i$, with a proper balance between the exploration of less known Ω_j and exploitation of promising Ω_j.

Following the construction of the classifier at each node, a trivial go-left greedy based search strategy, i.e. *Regression Tree* (RT), can be used to exclusively exploit the most promising Ω_k defined by the current action space. However, this is not good as manifested by the performance of RT in Fig. 6(c), and Fig. 7(d) (setting c = 0 degenerates UCB to RT), since it only *exploits* the current action space, which is learned from the current samples 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.

To overcome this issue, we use a variant of Monte Carlo Tree Search (MCTS) as the search method, which has the characteristics of adaptive exploration. Most importantly, MCTS has shown superior efficiency in high dimensional tasks, such as Go [13] and NAS [12]. MCTS keeps track of visitation statistics at each node to achieve the balance between the exploitation of existing good regions and the exploration of new regions. Like MCTS, our search phase also has *select*, *sampling* and *backpropagate* stages. LaNAS skips the *expansion* stage in regular MCTS since our search tree is static. When the action space is updated, previously sampled networks and their performance metrics in D_t are reused and redirected to (maybe different) nodes, when initializing visitation counts $n(s)$ and node values $v(s)$ for the tree with updated action space.

The search phase consists of 3 steps:

1) ***select w.r.t UCB***: the calculation of UCB follows the standard definition in [14]. We present the equation of π_{UCB} in Fig. 3. The input to π_{UCB} is the number of visits of current node $n(s)$ and next nodes $n(s,a)$, and the next node value $v(s,a)$, where s stands for a node, and a stands for an action. Therefore, the next node is deterministic given s and a. At node j, the number of visits $n(s)$ is determined by the number of samples in $D_t \cap \Omega_j$, while $v(s) = \hat{V}(\Omega_j)$. 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. **??** line 7-13).

2) ***sampling from a leaf***: *select* traverses a path from the root to a leaf, which defines a set of linear constrains for sampling. A node j defines a constraint l_j of $f_j(\mathbf{a}_i) \geq \hat{V}(\Omega_j), \forall \mathbf{a}_i \in \Omega$ if the path chooses the left child, and $f_j(\mathbf{a}_i) < \hat{V}(\Omega_j)$ otherwise. Therefore, the constraints from a path collectively enclose a partition Ω_j for proposing the new samples. Fig. 10 visualizes the process of partitioning along a search path.

Within a partition Ω_j, a simple search policy is to use reject sampling: random sample until it satisfies the constraints. This is efficient thanks to limited numbers of constraints [15], [16], [17]. Other strategies, e.g. Bayesian optimizations, can also be applied to sample from Ω_j. Here we illustrate the implementation of both π_{bayes} and π_{random}.

- *Random search based* π_{rand}: each component in the architecture encoding is assigned to a uniform random variable, and the vector of these random variables correspond to random architectures. For example, NASBench uses 21 Boolean variables for the adjacent matrix and 5 3-values categorical variables for the node list. The random generator uses 26 random integer variables, with 21 variables to be uniformly distributed in the set of [0, 1] indicating the existence of an edge, and 5 variables to be uniformly distributed in the set of [0, 1, 2] indicating layer types. π_{rand} outputs a random architecture as long as it satisfies the path constraints.
- *Bayesian search based* π_{bayes}: a typical Bayesian optimization search step consists of 2 parts: training a surrogate model using a Gaussian Process Regressor (GPR) on current collected samples D_t; and proposing new samples by optimizing the acquisition function, such as Expected Improvement (EI) or Upper Confidence Bound (UCB). However, GPR is not scalable to large samples; and we used meta-DNN in [3] to replace GPR. Training the surrogate remains unchanged on the same D_t collected by LaNAS. Inside the acquisition optimization, we only compute EI for architectures in the selected partition Ω_j, and returns $\mathbf{a}_i$ with the maximum EI. The search space of NAS is too large to be enumerable, we use a random generator to produce a pool of samples, e.g. 1 million, in Ω_j before computing EI. π_{bayes} outputs the sample with the largest EI out of the sample pool.

The comparisons of π_{bayes} to π_{random} is in Fig. 9. For the rest of the paper, we use π_{random} in LaNAS for the simplicity and strong final performance.

3) ***back-propagate reward***: after evaluating the sampled network, LaNAS back-propagates the reward, i.e. accuracy, to update the node statistics $n(s)$ and $v(s)$. It also back-propagates the sampled network so that every parent node j keeps the network in $D_t \cap \Omega_j$ for training.

There are multiple ways to evaluate the performance of architecture, such as training from scratch, or predicted from the shared weights as the case in one-shot NAS [18]. Among these methods, training every architecture from scratch (re-training) gives the most accurate v_i but is extremely costly. While one-shot NAS is fairly cheap to execute as it only requires one-time training of a supernet for predicting v_i for $\forall \mathbf{a}_i \in \Omega$, the predicted v_i is quite inaccurate [19]; therefore the architecture found by one-shot NAS is generally worse than the re-training approaches as indicated in Table. 2. In this paper, we try both one-shot based and re-training based evaluations. The integration of one-shot NAS is described as follows.

3.3 Integrating with one-shot NAS

The key bottleneck in NAS is evaluating samples by training a network from scratch. [18] proposes a weight sharing scheme to avoid the model re-training using a supernet, which can transform to any architectures in the search space

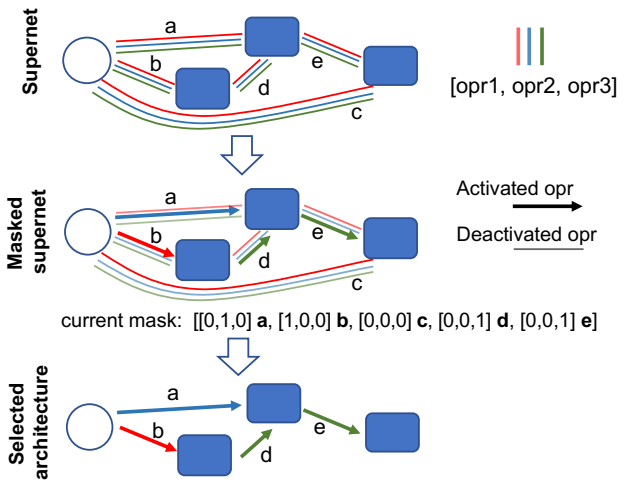
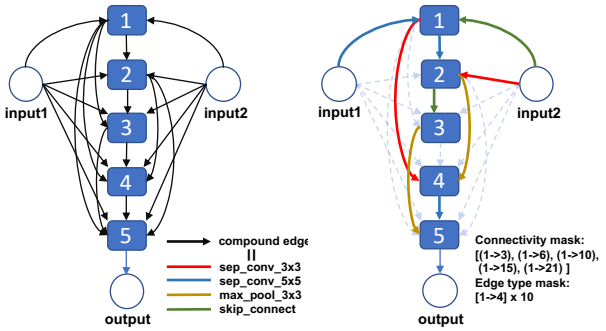


Fig. 4: **Integrating with one-shot NAS**: Before LaNAS comes into play, we pre-train a *supernet* with a random mask at each iteration until it converges, i.e. decoupling the training and search so that we can benchmark different algorithms on the same supernet. During the search phase, the supernet remains static. When LaNAS evaluate a network $\mathbf{a}_i$, we transform the supernet to $\mathbf{a}_i$ by multiplying the mask corresponding to $\mathbf{a}_i$ as the case shown in the figure. *Opr* stands for a layer type; we name edges from $a \rightarrow e$, and each edge can be one of the predefined layers or none. The figure shows there are 4 possibilities for an edge, represented either by a 1x3 one-hot vector to choose a layer type to activate the edge, or a 1x3 zero vector to deactivate the edge.



(a) supernet (b) masked supernet

Fig. 5: **The cell structure of supernet** used in searching nasnet. The supernet structure of normal and reduction cell are same. (a) Each edge is a compound edge, consisting of 4 independent edges with same input/output to represent 4 layer types. (b) Each nodes allows for two inputs from previous nodes. To specify a NASNet architecture, we use 5 variables for defining connectivity among nodes, and 10 variables for defining the layer type of every edges. Supernet can transform to any network in the search space by applying the mask.

by deactivating extra edges as shown in Fig. 4. One popular approach for one-shot NAS is to formulate the training of supernet and the search on supernet as an integrated bi-level optimizations [20], while recent works [21], [19] show this also can be done by separating the training and the search. Our work primarily focuses on the search efficiency, and separating the two procedures enables us to evaluate various algorithms on the same supernet (in Fig. 6(c)). We train the supernet by applying a random mask at each iteration following the same training pipeline/hyper-parameters in DARTs. After training the supernet, we fix the parameters of supernet, then evaluate various search methods onto it. For example, LaNAS samples $\mathbf{a}_i$, masking the supernet to evaluate $\mathbf{a}_i$ as illustrated by Fig. 4; then v_i is the validation accuracy evaluated from masked supernet. The result $(\mathbf{a}_i, v_i)$ is stored in D_t to guide the future search. The following elaborates the design of supernet for the NASNet search space, its training/search details and the masking process.

3.3.1 *The design of supernet*

We have used two designs of supernet in this paper: one is for NASNet search space [22] to be evaluated on CIFAR10, and the other is for EfficientNet search space [23] to be evaluated on ImageNet.

- *Supernet for NASNet search space*: our supernet follows the design of NASNet search space [22], the network of which is constructed by stacking multiple normal cells and reduction cells. Since the search space of normal/reduction cells is the same, the structure of supernet for both cells is also the same, shown in Fig. 5a. The supernet consists of 5 nodes, and each node connects to all previous nodes. While a NASNet only takes 2 inputs, we enforce this logic by masking. Each edge in supernet is a compound edge, consisting of 4 independent edges corresponding to 4 types of layers.
- *Supernet for EfficientNet search space*: we reused the supernet from [24], please refer to Once-For-All for details.

3.3.2 *Transforming supernet to a specific architecture*

There are two steps to transform a supernet to a target architecture by masking. Here we illustrate it on the NASNet search space, and the procedures on EfficientNet are same.

1) *Specifying an architecture*: The NASNet search space specifies two inputs to a node, which can be any previous nodes. Therefore, we used 5 integers to specify the connections of 5 nodes, and each integer enumerates all the possible connections of a node. For example, node 4 in Fig. 5a has 5 inputs, there are 5 possibilities $C(5,1)$ if two inputs are same, and 10 possibilities $C(5,2)$ for different inputs, adding up to 15 possible connections. Similarly, the possibilities for node1,2,3,5 are 3, 6, 10 and 21. Therefore, we use a vector of 5 integers with the range of $1 \rightarrow 3$, $1 \rightarrow 6$, $1 \rightarrow 10$, $1 \rightarrow 15$, and $1 \rightarrow 21$ to represent possible connections. After specifying the connectivity, we need to specify the layer type for each edge. In our experiments, a layer can be one of 3x3 separable convolution, 5x5 separable convolution, 3x3 max pooling and skip connect. Considering there are 10 edges in a NASNet cell, we use 10 integers ranging from 1 to 4 to represent the layer type chosen for an edge. Therefore, a NASNet can be fully specified with 15 integers (Fig. 5b).

2) *encoding to mask*: we need to change the encoding of 15 integers to a mask to deactivate the edges. Since the supernet in Fig 5a has 20 edges, we use a 20x4 matrix, with each row as a vector to specify a layer or deactivation. The conversion is straightforward; if an edge is activated in the encoding, the edge is a one-hot vector, or a vector of 0s otherwise.

3.3.3 Training supernet

As explained in sec. 3.3, we apply a random mask to each training iterations. We re-used the training pipeline from DARTs [20]. To generate the random mask, we used 15 random integers (explained in generating random masks above) to generate a random architecture with their ranges specified in Fig. 5b; then we transform the random encoding to a random mask, which is subsequently applied on supernet in training.

3.3.4 Searching on supernet

After training the supernet, it remains static for different algorithms to search on it. A search method proposes an architecture $\mathbf{a}_i$ for the evaluation; we mask the supernet to $\mathbf{a}_i$ by following the steps in Fig. 4, then evaluate the masked supernet to get v_i for $\mathbf{a}_i$. The new $\mathbf{a}_i$, v_i pair is stored in D_t to refine the search decision in the next iteration. Since the evaluation of $\mathbf{a}_i$ is reduced to evaluating masked supernet on the validation dataset, this has greatly reduced the computation cost, enabling a search algorithm to sample thousands of $\mathbf{a}_i$ in the reasonable amount of computations.

3.4 Partition Analysis

The sampling efficiency is closely related to the partition quality of each tree node. Here we seek an upper bound for the number of samples in the leftmost leaf (the most promising region) to characterize the sample efficiency. LaNAS shows more speedup w.r.t random search as the size of the search space grows. Details are in sec 3.4.

Assumption 1. Given a search domain Ω containing finite samples N, there exists a probabilistic density f such that $P(a < v < b) = \int_a^b f(v)dv$, where v is the performance of a network $\mathbf{a}$.

With this assumption, we can count the number of networks in the accuracy range of $[a, b]$ by $N * P(a \leq v \leq b)$. Since $v \in [0, 1]$ and $\sigma_(v) < \infty$, the following holds ([25])

$$|E(\overline{v} - M_v)| < \sigma_v \tag{3}$$

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

Theorem 1. Given a search tree of 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.

The theorem indicates that LaNAS is approximating the global optimum at the speed of $N/2^h$, suggesting 1) the performance improvement will remain near plateau as $h \uparrow$ (verified by Fig 7(a)), while the computational costs ($2^h - 1$ nodes) exponentially increase; 2) the performance improvement w.r.t random search (cost $\sim N/2$) is more obvious on a large search space (verified by Fig.5 (a)$\rightarrow$(c)).

Proof of Theorem: In the worst scenario, the left child is always assigned with the large 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_v; note $\delta \downarrow$ with more samples as $\epsilon \downarrow$, and σ_v becomes more accurate.

4 Experiment

4.1 Evaluating the search performance

4.1.1 Setup for benchmarks on NAS datasets

Choice of NAS datasets/supernet: NAS datasets record architecture-accuracy pairs for the fast retrieval by NAS algorithms to avoid time-consuming model retraining. This makes repeated runs of NAS experiments in a tractable amount of computing time to truly evaluate search algorithms. We use NASBench-101 [26] as one benchmark that contains over $4.2 * 10^5$ NASNet CNN models with edges $\leq$ 9 and nodes ≤ 7. To specify a network, search methods need 21 boolean variables for the adjacent matrix, and 5 3-value categorical variables for the node list[1], defining a search space of $|\Omega| = 5 * 10^8 \gg$ the size of dataset $|D| = 4.2 * 10^5$. In practice, NASBench returns 0 for the missing architectures, which potentially introduces a bias in evaluations. Besides, NASBench is still several orders of smaller than a search space in practice, e.g. NASNet [22] $|\Omega| \sim 10^{20}$. To resolve these issues, we curate a ConvNet dataset having $5.9 * 10^4$ samples to cover the case of $|D| = |\Omega|$, and a supernet with $|\Omega| = 3.5 * 10^{21}$ to cover the case of $\Omega \gg \Omega_{nasbench}$ for benchmarks.

The curation of ConvNet-60K follows similar procedures in collecting 1,364 networks in sec.2, free structural parameters that can vary are: network depth $D = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$, number of filters $C = \{32, 64, 96\}$ and kernel size $K = \{3 \times 3\}$, defining a $\Omega = 59049$. We train $\forall \mathbf{a}_i \in \Omega$ for 100 epochs, collect their final test accuracy v_i, store $(\mathbf{a}_i, v_i)$ in the dataset D. This small VGG-style, no residual connections, plain ConvNet search space can be fully specified with 10 3-value categorical variables, with each representing a type of filters.

We use a supernet on NASNet search space, and sec. 3.3 provides the details about the curation and the usage of a supernet in evaluating the search efficiency.

The architecture encoding: 1) *NASBench-101*: we used the architecture encoding of CIFAR-A in NASBench benchmarks from this repository[2], as the discrepancy between the size of the dataset and the search space is the minimal. Specifically, an architecture is encoded with 21 Boolean variables and 5 3-values categorical variables, with each value corresponding to 3 layer types, i.e. 3x3 convolution, 1x1 convolution, and max-pool. The 21 Boolean variables represent the adjacent matrix in NASBench, while the 5

1. this is the best encoding scheme with the minimal missing architectures, which is also used in NASBench baselines.
2. https://github.com/automl/nas_benchmarks

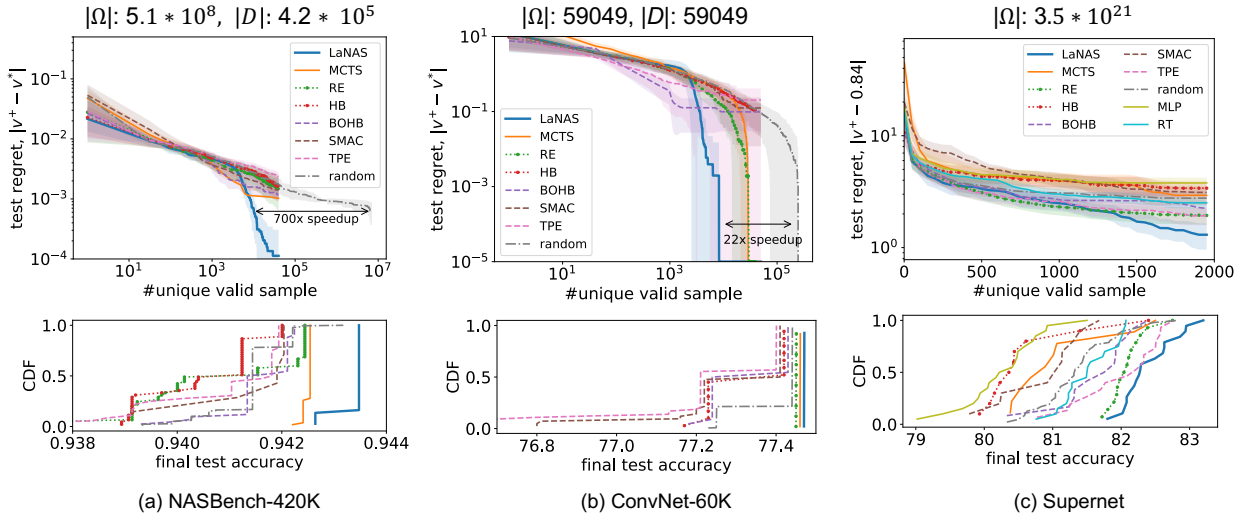


Fig. 6: The top row shows the time-course of test regrets of different methods (test regret between current best accuracy v^+ and the best in dataset v^* with the interquartile range), while the bottom row illustrates Cumulative Distribution Function (CDF) of v^+ for each method at $4 * 10^4$ unique valid samples. ConvNet-60K compensates NASBench to test the case of $|D| = |\Omega|$, and supernet compensates for the case of $|\Omega| \gg |\Omega_{nasbench}|$, where $|D|, |\Omega|$ are the size of the dataset and search space, respectively. LaNAS consistently demonstrates the best performance in 3 cases.

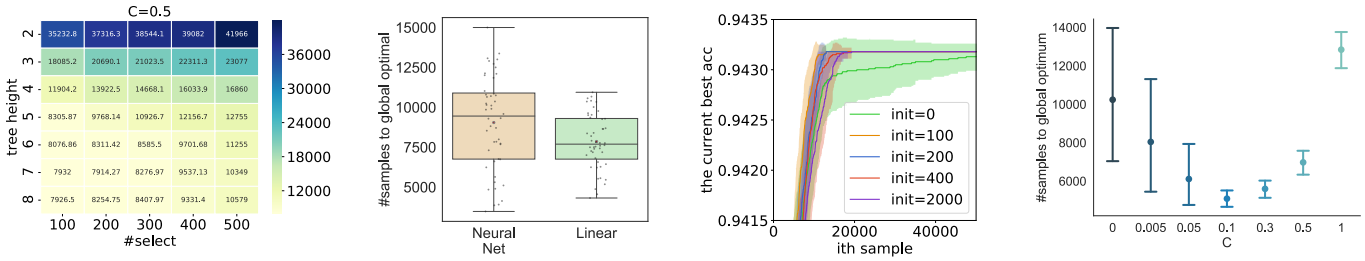


Fig. 7: **Ablation study**: (a) the effect of different tree heights and #select in MCTS. Number in each entry is #samples to reach global optimal. (b) the choice of predictor for splitting search space. (c) the effect of #samples for initialization toward the search performance. (d) the effect of C in UCB toward the performance on nasbench.

categorical variables represent the nodelist in NASBench. Therefore, $|\Omega| = 2^{21} * 3^5 = 5.1 * 10^8$. 2) *ConvNet-60K*: we used 10 3-values categorical variables to represent a VGG style CNN up to depth = 10, with each value correspond to 3 types of convolution layers , i.e. (filters=32, kernel = 3), (filters=64, kernel = 3) and (filters=96, kernel = 3). Therefore, $|\Omega| = 59049$. 3) *Supernet*: Since supernet implements the NASNet search space, the encoding of a supernet is same as the one used for NASBench-101.

Choice of baselines and setup for LaNAS: we adopt the same baselines established by NASBench-101, and the same implementations from this public release[3]. These baselines, summarized in Table. 1, cover diverse types of search algorithms. Regularized Evolution (RE) is a type of evolutionary algorithm that achieves SoTA performance for image recognition. While traditional BO method [27] suffers from the scalability issue (e.g. the computation cost scales $\mathcal{O}(n^3)$ with #samples), random forest-based Sequential Model-based Algorithmic Configuration (SMAC) and Tree of Parzen Estimators (TPE) are two popular solutions by using a scalable surrogate model. HyperBand (HB) is a resource-aware (e.g. training iterations or time) search method, and Bayesian optimization-based HyperBand (BOHB) extended HB for strong any time performance. In addition to baselines in NASBench-101, we also added MCTS to validate latent actions. We have extensively discussed LaNAS v.s. these baselines in sec 4.1.4.

In LaNAS, the height of the search tree is 8; we used 200 random samples for the initialization, and #select = 50 (the number of samples from a selected partition, see sec. 4.2).

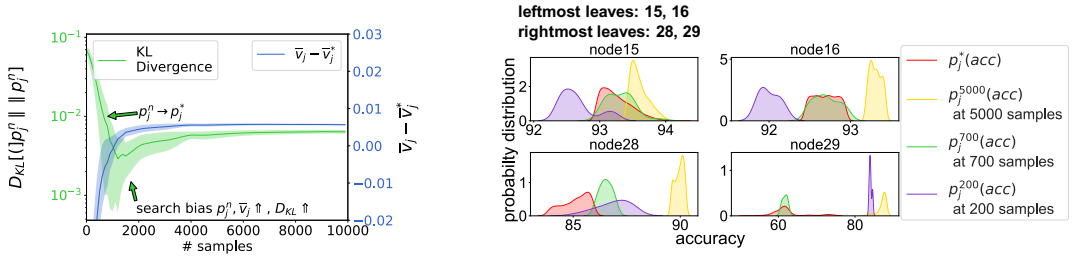
4.1.2 *Details about Ensuring Fairness*

1) *The encoding scheme*: the encoding scheme decides the size of the search space, thereby significantly affecting the performance. We ensure LaNAS and MCTS to use the same encoding as NASBench baselines on both datasets.

2) *Repeated samples*: we noticed NASBench baselines allow the same architecture to be sampled at different steps, and we modified LaNAS and MCTS to be consistent with baselines (by not removing samples from the search space).

3) *Evaluation metric*: we choose the number of unique,

3. https://github.com/automl/nas_benchmarks



(a) Search dynamics: KL-divergence and mean distance (b) Search dynamics: sample distribution vs. dataset distribution

Fig. 8: **Evaluations of search dynamics**:(a) 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. (b) 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 $n \in [700, 5000]$.

valid[4] samples instead of time to report the performance as model-free methods such as random search can easily iterate through the search space in a short time.

4) *Optimizing hyper-parameters*: the hyper-parameters of baselines are set w.r.t ablation studies in NASBench-101, and we also tuned LaNAS and MCTS for the benchmark.

5) *Sufficient runs with different random seeds*: each method is repeated 100 runs with different random seeds.

4.1.3 Empirical results

The top row of Fig. 6 shows the mean test regret, $|v^+ - v^*|$ where v^+ is the current best and v^* is the best in a dataset, along with the 25th and 75th percentile of each method through the course of searching, and the bottom row shows the robustness of methods at 40000 UVS on NASBench and ConvNet-60K, and 2000 UVS on supernet, respectively. Noted not all baselines are guaranteed to reach the global optimum, 40000 is the maximum UVS collected in 3 CPU days for all baselines on datasets, and 2000 is the maximum UVS on supernet in 3 GPU days.

We made the following observations: 1) *strong final performance*: LaNAS consistently demonstrates the strongest final performance on 3 tasks. On NASBench (Fig. 6(a)), the final test error of LaNAS is 0.011%, an order of smaller than the second best (0.137%); Similarly, on supernet (Fig. 6(c)), the highest test accuracy found by LaNAS is 83.5%, 0.75% better than the second best.

2) *good for one-shot NAS*: the strong final performance of LaNAS is more relevant to one-shot NAS as shown in Fig 6(c), as evaluations are fairly cheap.

3) *performance behavior*: across 3 experiments, the performance of LaNAS is comparable to Random Search in the few hundreds of samples ~ 500, and surpass baselines afterward. As an explanation for this behavior, we conduct a set of controlled experiments in Appendix. 4.3. We conclude that LaNAS needs a few hundreds of samples to accurately estimate boundaries, thereby good performance afterward.

4) *faster in larger* Ω: LaNAS is 700x and 22x faster than *random* in reaching similar regrets on NASBench-101 and ConvNet-60K. The empirical results validate our analysis (Appendix 3.4) that better performance w.r.t random search are observable on a larger search space.

4. we define valid samples as the samples in NASBench, and invalid as those in the search space but not in NASBench.

4.1.4 Discussions of baselines v.s. LaNAS

Like existing SMBO methods, LaNAS uses a tree of linear regressor as surrogate S in predicting the performance of unseen samples, and its S is proven to be quite effective as the resulting partitions clearly separate good/bad Ω_j (validated by Fig. 8(a), and Fig. 8(b) in Appendix 4.3). Besides, LaNAS uses π_{ucb} in MCTS as the acquisition to trade-off between exploration and exploitation; All together makes LaNAS more efficient than non-SMBO baselines. For example, RS relies on blind search, leading to the worst performance. RE utilizes a static exploration strategy that maintains a pool of top-K architectures for random mutations, not making full use of previous search experience. MCTS builds online models of both performance and visitation counts for adaptive exploration. However, without learning action space, the performance model at each node cannot be highly selective, leading to inefficient search (Fig. 2). The poor performance of HB attributes to the low-rank correlation between the performance at different budgets (Fig.7 in Supplement S2 of [26]).

Compared to Bayesian methods, LaNAS learns the state partitions to simplify the optimization of acquisition function ϕ. With learned actions, optimization is as simple as a quick traverse down the tree to arrive at the most performant region Ω_j, regardless the size of $|\Omega|$ and the dimensionality of tasks, and random sample a proposal within. Therefore, LaNAS gets a near optimal solution to $\max_{\mathbf{a}_i \in \Omega} \phi(\mathbf{a}_i)$ but without explicit optimization. In contrast, Bayesian methods such as SMAC, TPE and BOHB use iterated local search/evolutionary algorithm to propose a sample, which quickly becomes intractable on a high dimensional task, e.g. NAS with $|\Omega| > 10^{20}$. As a result, a sub-optimal solution to $\max_{\mathbf{a}_i \in \Omega} \phi(\mathbf{a}_i)$ leads to a sub-optimal sample proposal, thereby sub-optimal performance (shown in Fig 11, Appendix). Consistent with [28], our results in Fig. 6 also confirms it. For example, Bayesian methods, BOHB in particular, perform quite well w.r.t LaNAS on ConvNet (the low dimensional task) especially in the beginning when LaNAS has not learned its partitions well, but their relative performances dwindle on NASBench and supernet (high dimensional tasks) as the dimensionality

TABLE 2: Results on CIFAR-10, and c/o is cutout. Using ImageNet is either using ImageNet as the extra data or transfer weights from a network trained on ImageNet.

Model	Using ImageNet	Params	Top1 err	M	GPU days
search based methods					
NASNet-A+c/o [22]	X	3.3 M	2.65	20000	2000
AmoebaNet-B+c/o [10]	X	2.8 M	$2.55_{\pm 0.05}$	27000	3150
PNASNet-5 [29]	X	3.2 M	$3.41_{\pm 0.09}$	1160	225
NAO+c/o [30]	X	128.0 M	2.11	1000	200
AmoebaNet-B+c/o	X	34.9 M	$2.13_{\pm 0.04}$	27000	3150
EfficientNet-B7	√	64M	1.01		
BiT-M	√	60M	1.09		
LaNet+c/o	X	3.2 M	$\mathbf{1.63}_{\pm 0.05}$	800	150
LaNet+c/o	X	44.1 M	$\mathbf{0.99}_{\pm 0.02}$	800	150
one-shot NAS based methods					
ENAS+c/o [18]	X	4.6 M	2.89	-	0.45
DARTS+c/o [20]	X	3.3 M	$2.76_{\pm 0.09}$	-	1.5
BayesNAS+c/o [31]	X	3.4 M	$2.81_{\pm 0.04}$	-	0.2
ASNG-NAS+c/o [32]	X	3.9 M	$2.83_{\pm 0.14}$	-	0.11
XNAS+c/0 [33]	X	3.7 M	1.81		0.3
oneshot-LaNet+c/o	X	3.6 M	$\mathbf{1.68}_{\pm 0.06}$	-	3
oneshot-LaNet+c/o	X	45.3 M	$\mathbf{1.2}_{\pm 0.03}$	-	3

M: number of samples selected.

TABLE 3: Results on ImageNet.

Model	FLOPs	Params	top1 err
FBNetV2-C ([34])	375M	5.5 M	25.1
MobileNet-V3 ([])	219M	5.4 M	24.8
OFA ([24])	230M	5.1 M	23.1
FBNetV2-F4([35])	238M	5.6 M	24.0
LaNet	228M	5.1 M	**22.6**
NASNet-A ([22])	564M	5.3 M	26.0
AmoebaNet-C ([5])	570M	6.4 M	24.3
RandWire ([36])	583M	5.6 M	25.3
PNASNet-5 ([29])	588M	5.1 M	25.8
DARTS ([20])	574M	4.7 M	26.7
BayesNAS ([31])	-	3.9 M	26.5
OFA ([24])	595M	5.1 M	20.0
FBNetV3-([37])	544M	6.7 M	20.4
LaNet	598M	5.4 M	**19.2**

grows, $|\Omega_{supernet}| \gg |\Omega_{nasbench}| \gg |\Omega_{convnet}|$. Therefore, LaNAS is more effective than Bayesian methods in high-dimensional tasks.

4.2 Using LaNAS in practice

To be consistent with existing literature, we also evaluate LaNAS in searching for architectures for CIFAR-10 using the NASNet search space, and in searching for architectures for ImageNet using the EfficientNet search space.

In sec 4.1, LaNAS allows for repeated samples to be consistent with baselines, which is not desired in practice. Better performance are observable (Fig. 12 in Appendix) after removing existing samples D_t from search space Ω in optimizing the acquisition, i.e. $\max_{\mathbf{a}_i} \phi(\mathbf{a}_i), \forall \mathbf{a}_i \in \Omega \backslash D_t$, and LaNAS uses this logic through the rest of paper. Therefore, in the following ablation studies (sec. 4.2), LaNAS can find v^* on NASBench (Fig. 12) much faster than the case in Fig. 6.

On CIFAR-10, our search space is same as NASNet ([22]). We used operations of 3x3 max pool, 3x3, 5x5, depth-separable conv, and skip connection. The search target is the architectures for a reduction and a normal cell, and the number of nodes within a cell is 5. This formulates a search space of 3.5×10^{21} architectures. On ImageNet, our search space is consistent with Efficient-Net. The depth of a Inverted Residual Block (IRB) can be 2, 3, 4; and the expansion ration within a IRB can be 3, 5, 7. Therefore, the total possible architectures are around $10 * 10^{19}$.

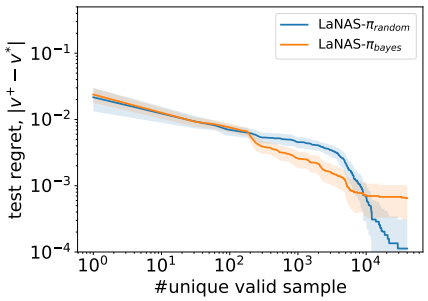


Fig. 9: Comparisons of π_{bayes} to π_{random} in sampling from the selected partition Ω_j

The setup of supernet on CIFAR-10 is consistent with the description in sec.3.3; on ImageNet, we reused supernet from [24]. We selected the top architecture collected from the search, and re-trained them for 600 epochs to acquire the final accuracy in Table. 2. We reused the training logic from `DARTS` and their training settings.

Table. 2 compares our results in the context of searching NASNet style architecture on CIFAR-10. The best performing architecture found by LaNAS in 800 samples demonstrates an average accuracy of 97.47% (#filters = 32, #params = 3.22M) and 98.01% (#filters = 128, #params = 38.7M), which is better than other results on CIFAR-10 without using ImageNet or transferring weights from a network per-trained on ImageNet. It is worth noting that we achieved this accuracy with 33x fewer samples than AmoebaNet. Besides, one-shot LaNAS also consistently demonstrates the strongest result among other one-shot variants in the similar GPU time. Besides, the results on ImageNet also consistently outperforms SoTA models.

The performance gap between one-shot NAS and search based NAS is largely due to inaccurate predictions of v_i from supernet [19], [38]. While improving supernet is beyond this work, we leave it as a future work.

4.2.1 *Hyper-parameters tuning in LaNAS*

The effect of tree height and #selects: Fig. 7(a) relates tree height (h) and the number of selects (#selects) to the search performance. Each entry represents #samples to find v^* 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 allowing the tree to make up-to-date decisions, and thus leads to improvement. On the other hand, the number of classifiers increases exponentially as the tree goes deeper, and a small #selects incurs frequent learning phase. Therefore, both can significantly increase the computation cost.

Choice of classifiers: Fig.7(b) shows that using a linear classifier performs better than an multi-layer perceptron (MLP). This indicates that adding complexity to decision boundary of actions may not help with the performance.

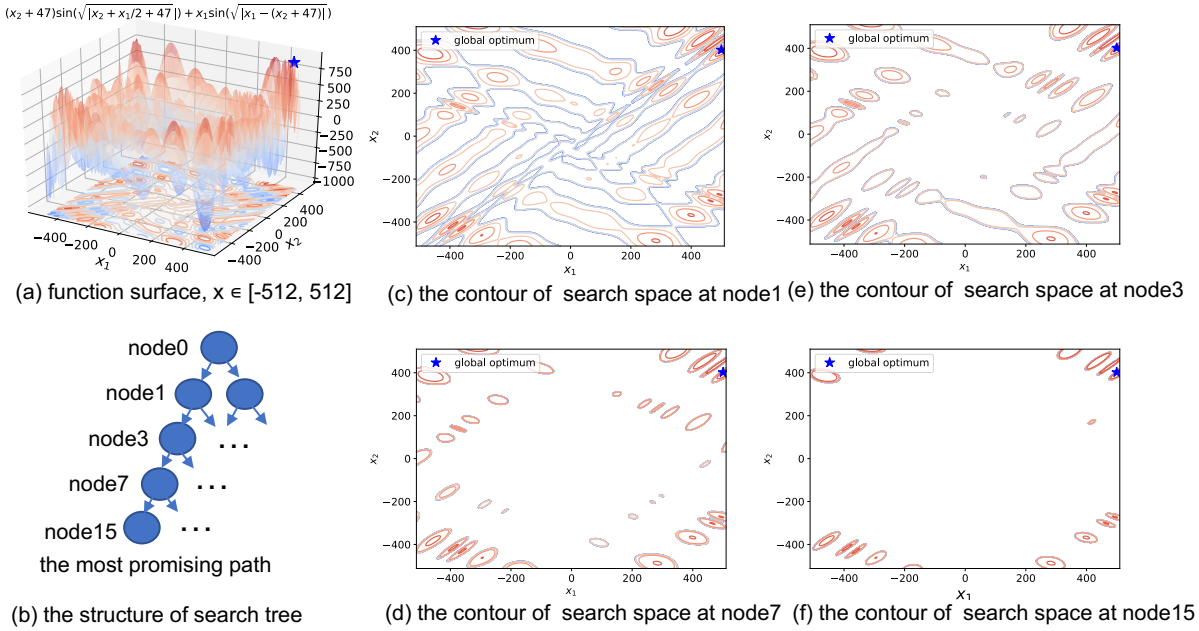


Fig. 10: **A visualization of partitioning eggholder function using LaNAS**: eggholder is a popular benchmark function for black-box optimization, (a) depicts its function surface, contour and definition. (b) LaNAS builds a tree of height = 5 for searching v^*; after collecting 500 samples, we visualize each partitions Ω_j represented at node0→node15 in (c)→(f), by splitting Ω based on its parent constraint. As node0→node15 recursively splitting Ω, the final Ω_j at node15 only contains the most promising region in Ω for sampling (see (f) v.s. (c)), and the bad region (blue lines in contours) are clearly separated from good region (red lines in contours) from (c)→(f).

Conversely, performance can get degraded due to potentially higher difficulties in optimization.

#samples for initialization: We need to initialize each node classifier properly with a few samples to establish the initial boundaries. Fig.7(c) shows cold start is necessary (init > 0 is better than init = 0), and a small init=100-400 converges to top 5% performance much faster than init=2000, while init=2000 gets the best performance a little faster.

The effect of c in UCB: Fig. 7(d) shows that the exploration term, $2c\sqrt{\frac{2log(n_{curt})}{n_{next}}}$, improves the performance as c increases from 0 to 0.1, while using a large c, e.g. > 0.5, is not desired for over-exploring. Please noted the optimal c is small as the maximum accuracy = 1. In practice, we find that setting c to $0.1*$max accuracy empirically works well. For example, if a performance metric in the range of [0, 100], we recommend setting c = 10.

Using π_{bayes} v.s. π_{random} for sample proposal: though π_{bayes} is faster in the beginning, π_{random} delivers the better final result due to the consistent random exploration in the most promising partition. Therefore, we used π_{random} for simplicity and good final performance through this paper.

4.3 Analysis of LaNAS

Experiment design: To validate the effectiveness of latent actions in partitioning the search space into regions with different performance metrics, and to visualize the search phase of LaNAS, we look into the dynamics of sample distributions on tree leaves during the search. By construction, left nodes contain regions of the good metric while the right nodes contain regions of the poor metric. Therefore, at each node j, we can construct *reference* distribution $p_j^*(v)$ by training toward a NAS dataset to partition the dataset into small regions with concentrated performances on leaves, i.e. using regression tree for the classification. Then, we compare $p_j^*(v)$ with the estimated distribution $p_j^n(v)$, where n is the number of accumulated samples in $D_t \cap \Omega_j$ at the node j at the search step t. Since the *reference* distribution $p_j^*(v)$ is static, visualizing $p_j^n(v)$ to $p_j^*(v)$ and calculating $D_{KL}[p_j^n \| p_j^*]$ enables us to see variations of the distribution over partition Ω_j on tree leaves w.r.t growing samples to validate the effectiveness of latent actions and to visualize the search.

Experiment setup: we used NASBench-101 that provides us with the true distribution of model accuracy, given any subset of model specifications, or equivalently a collection of actions (or constraints). 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, $\bar{v}_{15}^* > \bar{v}_{16}^* ... > \bar{v}_{29}^*$ reflected by $p_{15,16,28,29}^*$ in Fig. 8b.

Explanation to the performance of LaNAS: at the beginning of the search ($n = 200$ for random initialization), $p_{15,16}^{200}$ are expected to be smaller than $p_{15,16}^*$, and $p_{28,29}^{200}$ are expected to be larger than $p_{15,16}^*$; because the tree still learns to partition at that time. With more samples ($n = 700$), p_j starts to approximate p_j^*, manifested by the increasing similarity between $p_{15,16,28,29}^{700}$ and $p_{15,16,28,29}^*$, and the decreasing D_{KL} in Fig. 8a. This is because MCTS explores the under-explored regions, and it explains the comparable performance of LaNAS to baselines in Fig. 6. As the search continues ($n \to 5000$), LaNAS explores deeper

into promising regions and p_j^n is biased toward the region with good performance, deviated from p_j^*. As a result, D_{KL} bounces back in Fig. 8a. These search dynamics show how our model adapts to different stages during the course of the search, and validate its effectiveness in partitioning the search space.

Effective partitioning: The mean accuracy of $p_{15}^{700,5000} > p_{16}^{700,5000} > p_{28}^{700,5000} > p_{29}^{700,5000}$ in Fig. 8(b) indicates that LaNAS successfully minimizes the variance of rewards on a search path making architectures with similar metrics concentrated in a region, and LaNAS correctly ranks the regions on tree leaves. These manifest that LaNAS fulfills the online partitioning of Ω. An example partitioning of 2d egg-holder function can be found in Fig. 10.

5 Related works

Sequential Model Based Optimizations (SMBO) is a classic black box optimization framework [6], [39], that uses a surrogate S to extrapolate unseen region in Ω and to interpolate the explored region with existing samples. In the scenarios of expensive function evaluations $f(\mathbf{a}_i)$, SMBO is quite efficient by approximating $f(\mathbf{a}_i)$ with $S(\mathbf{a}_i)$. SMBO proposes new samples by solving $\max_{\mathbf{a}_i} \phi(\mathbf{a}_i)$ on S, where ϕ is a criterion, e.g. Expected Improvement (EI) [40] or Conditional Entropy of the Minimizer (CEM) [41], that transforms the value predicted from S for better trade-off between exploration and exploitation.

Bayesian Optimization (BO) is also an instantiation of SMBO [42], [43] that utilizes a Gaussian Process Regressor (GPR) as S. However, GPR suffers from the issue of $\mathcal{O}(n^3)$ where n is #samples. To resolve these issues, [39] replaces GPR with random forests, called SMAC-random forest, to estimate $\hat{\mu}$ and $\hat{\sigma}$ for predictive Gaussian distributions, and [9] proposes Tree-structured Parzen Estimator (TPE) in modeling Bayesian rule. Though both resolves the cubic scaling issue, as we thoroughly explained in sec 4.1.4, the key limitation of Bayesian approaches is at auxiliary optimization of acquisition function on an intractable search space. Similarly, recent predictor based methods [44] have achieved impressive results on NASBench by predicting every unseen architectures from the dataset. Without predicting over the entire dataset, their performance can drastically deteriorate. LaNAS eliminates this undesired constraint in BO or predictor based methods, being scalable regardless of problem dimensions, while still captures promising region for sample proposals.

Besides the recent success in games [45], [13], Monte Carlo Tree Search (MCTS) has also been used in robotics planning, optimization, and NAS [46], [47], [48], [49]. AlphaX is the first MCTS based NAS agent to explore the search space assisted with a value function predictor. However, the action space of AlphaX is manually defined w.r.t the search space. In sec. 2, we have clearly demonstrated that manually defined search space provides a confusing reward signal to the search, therefore low sample efficiency. In contrast, LaNAS learns the action space to partition the search space into good and bad regions, providing a stronger reward signal to guide the search; and [50] extends LaNAS to be a generic black box meta-solver.

6 Conclusion

This work presents a novel MCTS based search algorithm that learns action space for MCTS. With its application to NAS, LaNAS has proven to be more sample-efficient than existing approaches, validated by the cases with and without one-shot NAS on a diverse of tasks. The proposed algorithm is not limited to NAS, and has been extend to be a generic gradient-free algorithm [50], applied to different challenging black-box optimizations.

References

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

[2] B. Baker, O. Gupta, N. Naik, and R. Raskar, "Designing neural network architectures using reinforcement learning," *arXiv preprint arXiv:1611.02167*, 2016.

[3] L. Wang, Y. Zhao, Y. Jinnai, Y. Tian, and R. Fonseca, "Alphax: exploring neural architectures with deep neural networks and monte carlo tree search," *arXiv preprint arXiv:1903.11059*, 2019.

[4] S. Falkner, A. Klein, and F. Hutter, "Bohb: Robust and efficient hyperparameter optimization at scale," *arXiv preprint arXiv:1807.01774*, 2018.

[5] E. Real, A. Aggarwal, Y. Huang, and Q. V. Le, "Regularized evolution for image classifier architecture search," *arXiv preprint arXiv:1802.01548*, 2018.

[6] F. Hutter, "Automated configuration of algorithms for solving hard computational problems," Ph.D. dissertation, University of British Columbia, 2009.

[7] L. Li, K. Jamieson, G. DeSalvo, A. Rostamizadeh, and A. Talwalkar, "Hyperband: A novel bandit-based approach to hyperparameter optimization," *arXiv preprint arXiv:1603.06560*, 2016.

[8] F. Hutter, H. Hoos, and K. Leyton-Brown, "An evaluation of sequential model-based optimization for expensive blackbox functions," in *Proceedings of the 15th annual conference companion on Genetic and evolutionary computation*. ACM, 2013, pp. 1209–1216.

[9] J. S. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl, "Algorithms for hyper-parameter optimization," in *Advances in neural information processing systems*, 2011, pp. 2546–2554.

[10] E. Real, A. Aggarwal, Y. Huang, and Q. V. Le, "Regularized evolution for image classifier architecture search," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, 2019, pp. 4780–4789.

[11] L. Li and A. Talwalkar, "Random search and reproducibility for neural architecture search," *arXiv preprint arXiv:1902.07638*, 2019.

[12] L. Wang, Y. Zhao, Y. Jinnai, and R. Fonseca, "Alphax: exploring neural architectures with deep neural networks and monte carlo tree search," *arXiv preprint arXiv:1805.07440*, 2018.

[13] Y. Tian, J. Ma, Q. Gong, S. Sengupta, Z. Chen, J. Pinkerton, and C. L. Zitnick, "Elf opengo: An analysis and open reimplementation of alphazero," *arXiv preprint arXiv:1902.04522*, 2019.

[14] P. Auer, N. Cesa-Bianchi, and P. Fischer, "Finite-time analysis of the multiarmed bandit problem," *Machine learning*, vol. 47, no. 2-3, pp. 235–256, 2002.

[15] W. R. Gilks, N. G. Best, and K. Tan, "Adaptive rejection metropolis sampling within gibbs sampling," *Journal of the Royal Statistical Society: Series C (Applied Statistics)*, vol. 44, no. 4, pp. 455–472, 1995.

[16] W. Hörmann, "A rejection technique for sampling from t-concave distributions," *ACM Transactions on Mathematical Software (TOMS)*, vol. 21, no. 2, pp. 182–193, 1995.

[17] D. Görür and Y. W. Teh, "Concave-convex adaptive rejection sampling," *Journal of Computational and Graphical Statistics*, vol. 20, no. 3, pp. 670–691, 2011.

[18] H. Pham, M. Guan, B. Zoph, Q. Le, and J. Dean, "Efficient neural architecture search via parameter sharing," in *International Conference on Machine Learning*, 2018, pp. 4092–4101.

[19] C. Sciuto, K. Yu, M. Jaggi, C. Musat, and M. Salzmann, "Evaluating the search phase of neural architecture search," *arXiv preprint arXiv:1902.08142*, 2019.

[20] H. Liu, K. Simonyan, and Y. Yang, "Darts: Differentiable architecture search," *arXiv preprint arXiv:1806.09055*, 2018.

[21] Z. Guo, X. Zhang, H. Mu, W. Heng, Z. Liu, Y. Wei, and J. Sun, "Single path one-shot neural architecture search with uniform sampling," *arXiv preprint arXiv:1904.00420*, 2019.

[22] B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le, "Learning transferable architectures for scalable image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 8697–8710.

[23] M. Tan and Q. V. Le, "Efficientnet: Rethinking model scaling for convolutional neural networks," *arXiv preprint arXiv:1905.11946*, 2019.

[24] H. Cai, C. Gan, T. Wang, Z. Zhang, and S. Han, "Once-for-all: Train one network and specialize it for efficient deployment," *arXiv preprint arXiv:1908.09791*, 2019.

[25] C. Mallows, "Letters to the editor," *The American Statistician*, vol. 45, no. 3, pp. 256–262, 1991.

[26] C. Ying, A. Klein, E. Real, E. Christiansen, K. Murphy, and F. Hutter, "Nas-bench-101: Towards reproducible neural architecture search," *arXiv preprint arXiv:1902.09635*, 2019.

[27] J. Snoek, H. Larochelle, and R. P. Adams, "Practical bayesian optimization of machine learning algorithms," in *Advances in neural information processing systems*, 2012, pp. 2951–2959.

[28] Z. Wang, B. Shakibi, L. Jin, and N. de Freitas, "Bayesian multi- scale optimistic optimization," 2014.

[29] C. Liu, B. Zoph, M. Neumann, J. Shlens, W. Hua, L.-J. Li, L. Fei-Fei, A. Yuille, J. Huang, and K. Murphy, "Progressive neural architecture search," in *Proceedings of the European Conference on Computer Vision (ECCV)*, 2018, pp. 19–34.

[30] R. Luo, F. Tian, T. Qin, E. Chen, and T.-Y. Liu, "Neural architecture optimization," in *Advances in neural information processing systems*, 2018, pp. 7816–7827.

[31] H. Zhou, M. Yang, J. Wang, and W. Pan, "Bayesnas: A bayesian approach for neural architecture search," *arXiv preprint arXiv:1905.04919*, 2019.

[32] Y. Akimoto, S. Shirakawa, N. Yoshinari, K. Uchida, S. Saito, and K. Nishida, "Adaptive stochastic natural gradient method for one-shot neural architecture search," *arXiv preprint arXiv:1905.08537*, 2019.

[33] N. Nayman, A. Noy, T. Ridnik, I. Friedman, R. Jin, and L. Zelnik, "Xnas: Neural architecture search with expert advice," in *Advances in Neural Information Processing Systems*, 2019, pp. 1977–1987.

[34] B. Wu, X. Dai, P. Zhang, Y. Wang, F. Sun, Y. Wu, Y. Tian, P. Vajda, Y. Jia, and K. Keutzer, "Fbnet: Hardware-aware efficient convnet design via differentiable neural architecture search," *arXiv preprint arXiv:1812.03443*, 2018.

[35] A. Wan, X. Dai, P. Zhang, Z. He, Y. Tian, S. Xie, B. Wu, M. Yu, T. Xu, K. Chen *et al.*, "Fbnetv2: Differentiable neural architecture search for spatial and channel dimensions," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2020, pp. 12 965–12 974.

[36] S. Xie, A. Kirillov, R. Girshick, and K. He, "Exploring randomly wired neural networks for image recognition," *arXiv preprint arXiv:1904.01569*, 2019.

[37] X. Dai, A. Wan, P. Zhang, B. Wu, Z. He, Z. Wei, K. Chen, Y. Tian, M. Yu, P. Vajda *et al.*, "Fbnetv3: Joint architecture-recipe search using neural acquisition function," *arXiv preprint arXiv:2006.02049*, 2020.

[38] Y. Zhao, L. Wang, Y. Tian, R. Fonseca, and T. Guo, "Few-shot neural architecture search," *arXiv preprint arXiv:2006.06863*, 2020.

[39] F. Hutter, H. H. Hoos, and K. Leyton-Brown, "Sequential model-based optimization for general algorithm configuration," in *International conference on learning and intelligent optimization*. Springer, 2011, pp. 507–523.

[40] D. R. Jones, "A taxonomy of global optimization methods based on response surfaces," *Journal of global optimization*, vol. 21, no. 4, pp. 345–383, 2001.

[41] J. Villemonteix, E. Vazquez, and E. Walter, "An informational approach to the global optimization of expensive-to-evaluate functions," *Journal of Global Optimization*, vol. 44, no. 4, p. 509, 2009.

[42] Z. Wang, M. Zoghi, F. Hutter, D. Matheson, and N. De Freitas, "Bayesian optimization in high dimensions via random embeddings," in *Twenty-Third International Joint Conference on Artificial Intelligence*, 2013.

[43] J. R. Gardner, M. J. Kusner, Z. E. Xu, K. Q. Weinberger, and J. P. Cunningham, "Bayesian optimization with inequality constraints." in *ICML*, 2014, pp. 937–945.

[44] H. Shi, R. Pi, H. Xu, Z. Li, J. T. Kwok, and T. Zhang, "Multi-objective neural architecture search via predictive network performance optimization," *arXiv preprint arXiv:1911.09336*, 2019.

[45] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot *et al.*, "Mastering the game of go with deep neural networks and tree search," *nature*, vol. 529, no. 7587, p. 484, 2016.

[46] L. Buşoniu, A. Daniels, R. Munos, and R. Babuška, "Optimistic planning for continuous-action deterministic systems," in *2013 IEEE Symposium on Adaptive Dynamic Programming and Reinforcement Learning (ADPRL)*. IEEE, 2013, pp. 69–76.

[47] R. Munos, "From bandits to monte-carlo tree search: The optimistic principle applied to optimization and planning," *technical report*, vol. x, no. x, p. x, 2014.

[48] A. Weinstein and M. L. Littman, "Bandit-based planning and learning in continuous-action markov decision processes," in *Twenty-Second International Conference on Automated Planning and Scheduling*, 2012.

[49] C. Mansley, A. Weinstein, and M. Littman, "Sample-based planning for continuous action markov decision processes," in *Twenty-First International Conference on Automated Planning and Scheduling*, 2011.

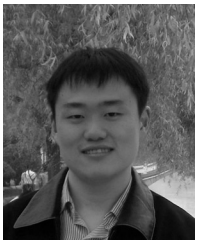
[50] L. Wang, R. Fonseca, and Y. Tian, "Learning search space partition for black-box optimization using monte carlo tree search," *Advances in Neural Information Processing Systems*, 2020.

Linnan Wang Linnan is a Ph.D. student at the CS department of Brown University, advised by Prof.Rodrigo Fonseca. Before Brown, he was a OMSCS student at Gatech while being a full time software developer at Dow Jones. He acquired his bachelor degree from University of Electronic Science and Technology of China (UESTC). His research interests include Artificial Intelligence (AI) and High Performance Computing (HPC).

Saining Xie Saining is a research scientist at Facebook AI Research. He was a PhD student at CSE Department, UC San Diego, advised by Professor Zhuowen Tu. He obtained his bachelor degree from Shanghai Jiao Tong University in 2013. His research interest includes machine learning (especially deep learning) and its applications in computer vision.

Teng Li Teng is a a research scientist at Facebook AI Research. He received my PhD in computer engineering from the George Washington University in Washington, DC. His PhD research is primarily within the area of GPGPU, parallel and distributed systems, and high-performance computing.

Rodrigo Fonseca Rodrigo is an associate professor at Brown University's Computer Science Department. His work revolves around distributed systems, networking, and operating systems. Broadly, He is interested in understanding the behavior of systems with many components for enabling new functionality, and making sure they work as they should.



Yuandong Tian Yuandong is a Research Scientist in Facebook AI Research (FAIR). He received Ph.D in the Robotics Institute, Carnegie Mellon University, advised by Srinivasa Narasimhan. He received Master and Bachelor degrees in Computer Science and Engineering Department, Shanghai Jiao Tong University. He is a recipient of ICCV 2013 Marr Prize Honorable Mentions for a hierarchical framework that gives globally optimal guarantees for non-convex non-rigid image deformation.

7 APPENDIX

7.1 LaNAS algorithms

Algorithm 1 LaNAS search procedures

while acc $<$ target **do**
  for $n \in Tree.N$ **do**
    $n.g.train()$
  end for
  for $i = 1 \rightarrow \#selects$ **do**
    $leaf, path = ucb\_select(root)$
    $constraints = get\_constraints(path)$
    $network = sampling(constraints)$
    $acc = network.train()$
    $back\_propagate(network, acc)$
  end for
end while

Algorithm 2 get_constraints(path) in Alg. 1

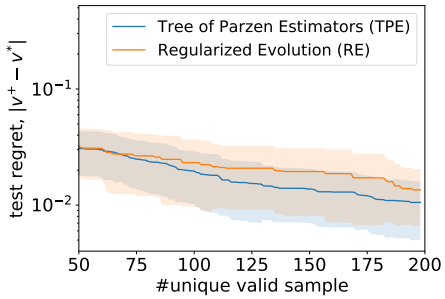
$constraints$ = []
for $node \in s\_path$ **do**
  $\mathbf{W}, b = node.g.params()$
  $\bar{X} = node.\bar{X}$
  if $node$ on left **then**
    $constraints.add(\mathbf{W}\mathbf{a} + b \geq \bar{X})$
  else
    $constraints.add(\mathbf{W}\mathbf{a} + b < \bar{X})$
  end if
end for
return $constraints$

Algorithm 3 ucb_select($c = root$) in Alg. 1

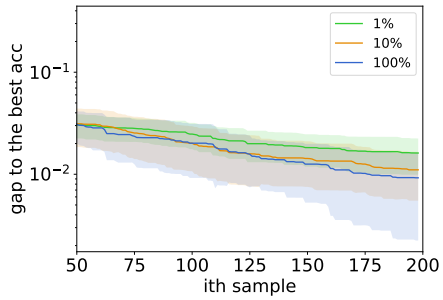
$path = []$
while c not $leaf$ **do**
  $path.add(c)$
  $l_{ucb} = get\_ucb(c.left.\bar{X}, c.left.n, c.n)$
  $r_{ucb} = get\_ucb(c.right.\bar{X}, c.right.n, c.n)$
end while
while c not $leaf$ **do**
  $path.add(c)$
  if $l_{ucb} > r_{ucb}$ **then**
    $c = c.left$
  else
    $c = c.right$
  end if
end while
return $path, c$

Algorithm 4 get_ucb($\bar{X}_{next}$, n_{next}, n_{curt}) in Alg. 3

c = 0.1
if $n_{next} = 0$ **then**
  return $+\infty$
else
  return $\frac{\bar{X}_{next}}{n_{next}} + 2c\sqrt{\frac{2log(n_{curt})}{n_{next}}}$
end if



(a) max$\phi(\mathbf{a}_i)$ using TPE and RE



(b) max$\phi(\mathbf{a}_i)$ with the hill climbing using 1%, 10% and 100% of search space Ω

Fig. 11: **The impact of max$\phi(\mathbf{a}_i)$ in the Bayesian Optimization (BO)**: BO proposes samples by a non-convex optimization of the acquisition function, i.e. max$\phi(\mathbf{a}_i)$. **(a) shows** that the BO performance is closely related to the performance of search methods used in $\max_{\mathbf{a}_i}\ \phi(\mathbf{a}_i)$, and **(b) shows** the BO performance deteriorates after reducing the search budget for a local search algorithm (hill climbing) from probing 100% search space Ω to, probing 1% of Ω in max$\phi(\mathbf{a}_i)$. In a high dimensional search space, e.g. NAS $|\Omega| \sim 10^{20}$, it is impossible to find the global optimum in max$\phi(\mathbf{a}_i)$, thereby deteriorating the performance of BO methods.

7.2 Implementing different sample selection policies on the selected partition Ω_j

7.3 Setup for evaluating the search performance

7.3.1 Setup for benchmarks on supernet

7.4 Experiment setup for :

The best convolutional cell found by LaNAS is visualized in Fig. 13. For details of constructing a network with learned cells, please refer to Fig.15 in [3].

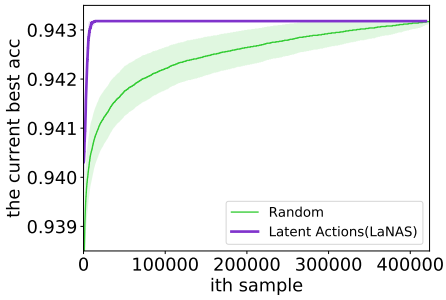
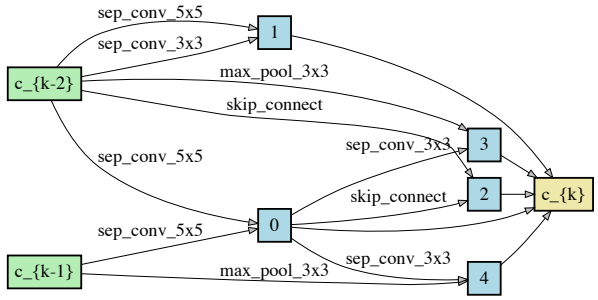
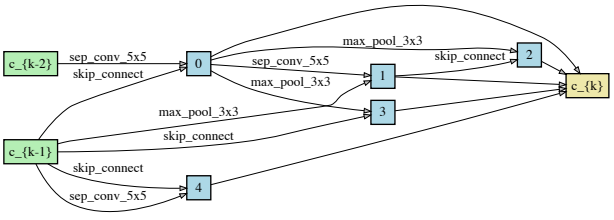


Fig. 12: **The performance on NASBench** by $\max_{\mathbf{a}_i} \phi(\mathbf{a}_i)$, $\forall \mathbf{a}_i \in \Omega - D_t$: in sec. 4.1, we propose new samples by $\max_{\mathbf{a}_i} \phi(\mathbf{a}_i), \forall \mathbf{a}_i \in \Omega$ to be consistent with baselines for fair comparisons. However, this is not desired in practice, and better performance are observable after removing current samples D_t from Ω in optimizing the acuquistion ϕ, i.e. $\Omega - D_t$. On NASBench, LaNAS can quickly find the best architecture in the dataset after this modification; therefore LaNAS uses $\Omega - D_t$ for sample proposal through this paper except for sec . 4.1.



(a) normal cell



(b) reduction cell

Fig. 13: **searched cell structure**: the best cell structure found in the search.