AlphaX: eXploring Neural Architectures with Deep Neural Networks and Monte Carlo Tree Search

Linnan Wang[1] Yiyang Zhao Yuu Jinnai[1] Yuandong Tian[2] Rodrigo Fonseca[1]
[1]Brown University [2]Facebook AI Research

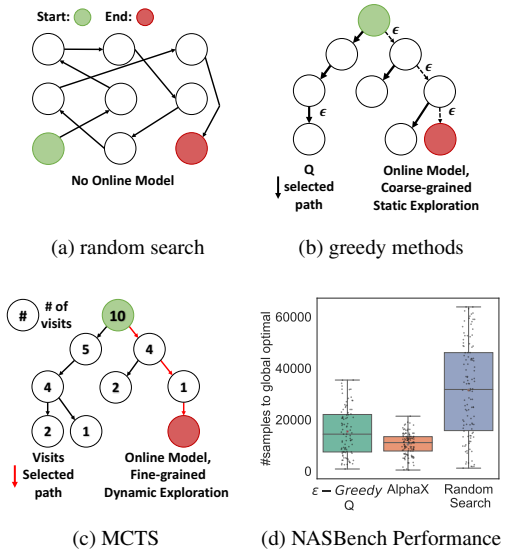
Abstract

We present AlphaX, a fully automated agent that designs complex neural architectures from scratch. AlphaX explores the search space with a distributed Monte Carlo Tree Search (MCTS) and a Meta-Deep Neural Network (DNN). MCTS guides transfer learning and intrinsically improves the search efficiency by dynamically balancing the exploration and exploitation at fine-grained states, while Meta-DNN predicts the network accuracy to guide the search, and to provide an estimated reward to speed up the rollout. As the search progresses, AlphaX also generates the training data for Meta-DNN. So, the learning of Meta-DNN is end-to-end. In 8 GPU days, AlphaX found an architecture that reaches 97.88% top-1 accuracy on CIFAR-10, and 75.5% top-1 accuracy on ImageNet. We also evaluate AlphaX on a large scale NAS dataset for reproducibility. On NASBench-101, AlphaX also demonstrates 3x and 2.8x speedup over Random Search and Regularized Evolution in finding the global optimum. Finally, we show the searched architecture improves a variety of vision applications from Neural Style Transfer, to Image Captioning and Object Detection.

1. Introduction

Designing efficient neural architectures is extremely laborious. A typical design iteration starts with a heuristic design hypothesis from domain experts, followed by the design validation with hours of GPU training. The entire design process renders many of such iterations before finding a satisfying architecture. Neural Architecture Search (NAS) has emerged as a promising tool to alleviate human effort in this trial and error design process, but the tremendous computing resources required by current NAS methods motivates us to investigate the search efficiency.

AlphaGo/AlphaGoZero [33] recently show super-human performance in playing the game of Go, by using a specific search algorithm called Monte-Carlo Tree Search (MCTS)[14, 4]. Given the current game state, MCTS gradually builds an online model for its subsequent game states to evaluate the winning chance at that state, based on search experiences in the current and prior games, and makes a decision. Search experience is from the previous search trajectories (called *rollouts*) that have been tried, and their consequences (whether the player wins or not). Different from traditional MCTS approach that evaluates the consequence of a trajectory by random self-play to the end of a game, AlphaGo uses a *predictive model* (or value network) to predict the consequence, which enjoys much lower variance. Furthermore, due to its built-in exploration mechanism using *Upper Confidence bound applied to Trees* (UCT)[2], based



(a) random search (b) greedy methods

(c) MCTS (d) NASBench Performance

Figure 1: **Comparisons of NAS algorithms**: **(a)** *random search* makes independent decision without using prior rollouts (previous search trajectories). An online model is to evaluate how promising the current search branch based on prior rollouts, and Random Search has no online model. **(b)** Search methods guided by online performance models built from previous rollouts. With static, coarse-grained exploration strategy (e.g., ϵ-greedy in *Q-learning*), they may quickly be stuck in a sub-optimal solution; and the chance to escape is exponentially decreasing along the trajectory. **(c)** AlphaX builds online models of both performance and visitation counts for adaptive exploration. **(d)** Performance of different search algorithms on NASBench-101 [43]. AlphaX is 3x, 1.5x more sample-efficient than *random search* and ϵ-*greedy* based Q learning.

Methods	Global Solution	Online Model	Exploration	Distributed Ready	Transfer Learning
MetaQNN (QL) [3]	×	-	ϵ-greedy	×	×
Zoph (PG)[44]	×	RNN	-	√	×
PNAS (HC)[17]	×	RNN	-	√	×
Regularized Evolution[30]	√	-	top-k mutation	√	×
Random Search[32]	√	-	random	√	×
DeepArchitect (MCTS)[28]	√	-	UCT	×	×
Wistuba (MCTS)[40]	√	Gaussian	UCT	×	×
AlphaX (MCTS)	√	meta-DNN	UCT	√	√

Table 1: **Highlights of NAS search algorithms**: compared with DeepArchitect and Wistuba, AlphaX features a scalable online model, and is ready for distributed system.

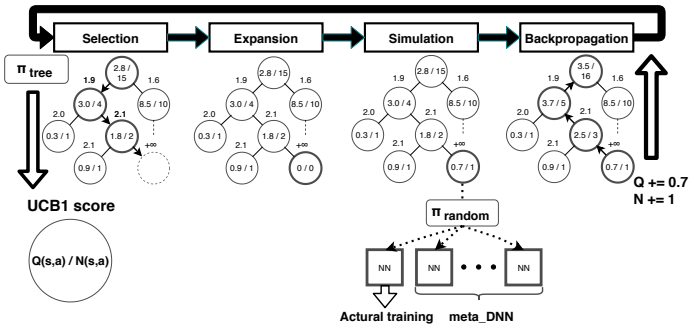


Figure 2: **Overview of AlphaX search procedures:** explanations of four steps are in sec.3.2.

on its online model, MCTS dynamically adapts itself to the most promising search regions, where good consequences are likely to happen.

Inspired by this idea, we present AlphaX that uses MCTS for efficient model architecture search with Meta-DNN as a predictive model to estimate the accuracy of a sampled architecture. Compared with Random Search, AlphaX builds an online model which guides the future search, compared to greedy methods, e.g. Q-learning, Regularized Evolution or Top-K methods, AlphaX dynamically trades off exploration and exploitation and can escape from locally optimal solutions with fewer number of search trials. Fig. 1 summarizes the trade-offs. Furthermore, while prior works on MCTS applied to Architecture Search [40, 28] only report performance on CIFAR-10, our distributed AlphaX system can scale up to 32 machines (with 32 acceleration ratio), reports better accuracies on CIFAR-10 and achieves performance on large-scale dataset like ImageNet that is on par with the SOTA. Particularly, AlphaX is up to 3x faster than Random Search and Regularized Evolution in finding the best performing network on NASBench-101.

2. Related Work

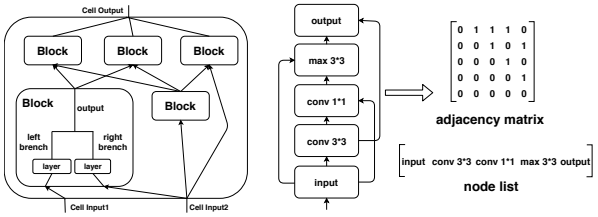
Monte Carlo Tree Search: DeepArchitect[28] implemented vanilla MCTS for NAS without a predictive model, and Wistuba [40] uses statistics from the current search (e.g., RAVE and Contextual Reward Prediction) to predict performance of a state. In comparison, our performance estimate is from both searched rollouts so far and a model (meta-DNN) learned from performances of known architectures and can generalize to unseen architectures. Both previous works report performance on CIFAR-10, while AlphaX reports performance on CIFAR-10, ImageNet, and NASBench-101. Most importantly, AlphaX is the first scalable MCTS based design agent.

Bayesian Optimization (BO) is a popular method for the hyper-parameter search [11, 34, 39, 13]; It is proven to be an effective black-box optimization technique for small scale problems, e.g. finding good hyper-parameters for Stochastic Gradient Descent (SGD). In a large-scale problem, it demands a good, sophisticated high-dimensional representation kernel to work, requires calculating the inverse of a covariance matrix $\mathcal{O}(n^{2.376})$ that quadratically increases with samples (n), so BO is not a practical algorithm for NAS.

Reinforcement Learning (RL): Several RL techniques have been investigated for NAS [3, 44]. Baker et al. proposed a Q-learning agent to design network architectures [3]. The agent takes a ϵ-greedy policy: with probability $1-\epsilon$, it chooses the action that leads to the best expected return (i.e. accuracy) estimated by the current model, otherwise uniformly chooses an action. Zoph et al. built an RNN agent trained with Policy Gradient to design CNN and LSTM [44]. However, directly maximizing the expected reward in vanilla Policy Gradient could lead to local optimal solution [27]. In comparison, MCTS records both expected return and visitation counts of states to balance between exploration and exploitation at the state level.

Hill Climbing (HC): Elsken et al. proposed a simple hill climbing for NAS [7]. Starting from an architecture, they train every descendent network before moving to the best performing child. Liu et al. deployed a beam search which follows a similar procedure to hill climbing but selects the top-K architectures instead of only the best [17]. HC is akin to the vanilla Policy Gradient tending to trap into a local optimum from which it can never escape, while MCTS demonstrates provable convergence toward the global optimal given enough time[14].

Evolutionary Algorithm (EA): Evolutionary algorithms represent each neural network as a string of genes and search the architecture space by mutation and recombinations [26, 36, 35, 12, 8, 19, 31, 41, 25, 37, 30]. Strings which represent a neural network with top performance are selected to generate child models. The selection process is in lieu of exploitation, and the mutation is to encourage exploration. Still, GA algorithms do not consider the visiting statistics at individual states, and its performance is on par with *random search* for lacking an online model to inform decisions.



(a) NASNet Cell (b) NASBench DAG

Figure 3: **Design space**: (a) the cell structure of NASNet and (b) the DAG structure of NASBench-101. Then the network is constructed by stacking multiple Cells or DAGs.

3. AlphaX: A Scalable MCTS Design Agent

3.1. Design, State and Action Space

Design Space: the neural architectures for different domain tasks, e.g. the object detection and the image classification, follow fundamentally different designs. This renders different design spaces for the design agent. AlphaX is flexible to support various search spaces with an intuitive state and action abstraction. Here we provide a brief description of two search spaces used in our experiments.

- *NASNet Search Space*: [45] proposes searching a hierarchical Cell structure as shown in Fig.3a. There are two types of Cells, Normal Cell ($NCell$) and Reduction Cell ($RCell$). Normal Cell maintains the input and output dimensions with the padding, while Reduction Cell reduces the height and width by half with the striding. Then, the network is constituted by stacking multiple cells.

- *NASBench Search Space*: [43] proposes searching a small Direct Acyclic Graph (DAG) with each node representing a layer and each edge representing the inter-layer dependency. Similarly, the network is constituted by stacking multiple such DAGs.

State Space: a state represents a network architecture, and AlphaX utilizes states (or nodes) to keep track of past trails to inform future decisions. We implement a state as a map that defines all the hyper-parameters for each network layers and their dependencies. We also introduce a special terminal state to allow for multiple actions. All the other states can transit to the terminal state by taking the terminal action, and the agent only trains the network, from which it reaches the terminal. With the terminal state, the agent freely modifies the architecture before reaching the terminal. This enables multiple actions for the design agent to bypass shallow architectures.

Action Space: an action morphs the current network architecture, i.e. current state, to transit to the next state. It not only explicitly specifies the inter-layer connectivity, but also all the necessary hyper-parameters for each layer. Unlike games, actions in NAS are dynamically changing w.r.t the current state and design spaces. For example, AlphaX needs to leverage the current DAG (state) in enumerating all the feasible actions of 'adding an edge'. In our experiments, the actions for the NASNet search domain are adding a new layer in the left or right branch of a $Block_i$ in a $Cell$, creating a new $Block$ with different input combinations, and the terminating. The actions for the NASBench search domain are either adding a node or an edge, and the terminating.

3.2. Search Procedure

This section elaborates the integration of MCTS and metaDNN in AlphaX. The MCTS is to analyze the most promising move at a state, while the meta-DNN is to learn the sampled architecture performance and to generalize to unexplored architectures so that MCTS can simulate many rollouts with only an actual network training in evaluating a new node. The superior search efficiency of AlphaX is due to balancing the exploration and exploitation at the finest granularity, i.e. state level, by leveraging the visiting statistics. Each node tracks these two statistics: 1) $N(s,a)$ counts the selection of action a at state s; 2) $Q(s,a)$ is the expected reward after taking action a at state s, and intuitively $Q(s,a)$ is an estimate of how promising this search direction. Fig.2 demonstrates a typical searching iteration in AlphaX, which consists of *Selection*, *Expansion*, *Meta-DNN assisted Simulation*, and *Backpropagation*. We elucidate each step as follows.

Selection traverses down the search tree to trace the current most promising search path. It starts from the root and stops till reaching a leaf. At a node, the agent selects actions based on UCB1 [2]:

$$\pi_{tree}(s) = \arg\max_{a \in A} \left(\frac{Q(s,a)}{N(s,a)} + 2c\sqrt{\frac{2\log N(s)}{N(s,a)}} \right), \quad (1)$$

where $N(s)$ is the number of visits to the state s (i.e. $N(s) = \sum_{a \in A} N(s,a)$), and c is a constant. The first term ($\frac{Q(s,a)}{N(s,a)}$) is the exploitation term estimating the expected accuracy of its descendants. The second term ($2c\sqrt{\frac{2\log N(s)}{N(s,a)}}$) is the exploration term encouraging less visited nodes. The exploration term dominates $\pi_{tree}(s)$ if $N(s,a)$ is small, and the exploitation term otherwise. As a result, the agent favors the exploration in the beginning until building proper confidences to exploit. c controls the weight of exploration, and it is empirically set to 0.5. We iterate the tree policy to reach a new node.

Expansion adds a new node into the tree. $Q(s,a)$ and $N(s,a)$ are initialized to zeros. $Q(s,a)$ will be updated in the simulation step.

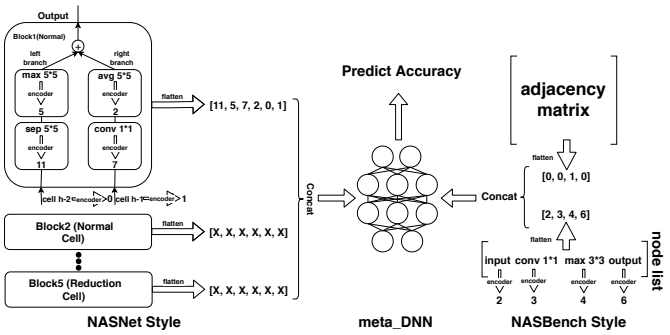


Figure 4: Encoding scheme of NASBench and NASNet.

Meta-DNN assisted Simulation randomly samples the descendants of a new node to approximate $Q(s,a)$ of the node with their accuracies. The process is to estimate how promising the search direction rendered by the new node and its descendants. The simulation starts at the new node. The agent traverses down the tree by taking the uniform-random action until reaching a terminal state, then it dispatches the architecture for training.

The more simulation we roll, the more accurate estimate of this search direction we get. However, we cannot conduct many simulations as network training is extremely time-consuming. AlphaX adopts a novel hybrid strategy to solve this issue by incorporating a meta-DNN to predict the network accuracy in addition to the actual training. We delay the introduction of meta-DNN to sec.3.3. Specifically, we estimate $q = Q(s,a)$ with

$$Q(s,a) \leftarrow \left(Acc(sim_0(s')) + \frac{1}{k} \sum_{i=1..k} Pred(sim_i(s')) \right) / 2 \tag{2}$$

where $s' = s + a$, and $sim(s')$ represents a simulation starting from state s'. Acc is the actually trained accuracy in the first simulation, and $Pred$ is the predicted accuracy from Meta-DNN in subsequent k simulations. If a search branch renders architectures similar to previously trained good ones, Meta-DNN updates the exploitation term in Eq.1 to increase the likelihood of going this branch.

Backpropagation back-tracks the search path from the new node to the root to update visiting statistics. Please note we discuss the sequential case here, and the backpropagation will be split into two parts in the distributed setting (sec.3.5). With the estimated q for the new node, we iteratively back-propagate the information to its ancestral as:

$$\begin{aligned} &Q(s,a) \leftarrow Q(s,a) + q, \quad N(s,a) \leftarrow N(s,a) + 1 \\ &s \leftarrow parent(s), \quad a \leftarrow \pi_{tree}(s) \end{aligned} \tag{3}$$

until it reaches the root node.

3.3. The design of Meta-DNN and its related issues

Meta-DNN intends to generalize the performance of unseen architectures based on previously sampled networks. It provides a practical solution to accurately estimate a search branch with many simulations without involving the actual training (see the metaDNN assisted simulation for details). New training data is generated as AlphaX advances in the search. So, the learning of Meta-DNN is end-to-end. The input of Meta-DNN is a vector representation of architecture, while the output is the prediction of architecture performance, i.e. test accuracy.

The coding scheme for NASNet architectures is as follows: we use 6-digits vector to code a $Block$; the first two digits represent up to two layers in the left branch, and the 3rd and 4th digits for the right branch. Each layer is represented by a number in $[1, 12]$ to represent 12 different layers, and the specific layer code is available in Appendix TABLE.4. We use 0 to pad the vector if a layer is absent. The last two digits represent the input for the left and right branch, respectively. For the coding of block inputs, 0 corresponds to the output of the previous $Cell$, 1 is the previous, previous $Cell$, and $i + 2$ is the output of $Block_i$. If a block is absent, it is [0,0,0,0,0,0]. The left part of Fig.4 demonstrates an example of NASNet encoding scheme. A $Cell$ has up to 5 blocks, so a vector of 60 digits is sufficient to represent a state that fully specifies both $RCell$ and $NCell$. The coding scheme for NASBench architectures is a vector of flat adjacency matrix, plus the nodelist. Similarly, we pad 0 if a layer or an edge is absent. The right part of Fig.4 demonstrates an example of NASBench encoding scheme. Since NASBench limits nodes ≤ 7, 7×7 (adjacency matrix)+ 7 (nodelist) = 56 digits can fully specify a NASBench architecture.

Now we cast the prediction of architecture performance as a regression problem. Finding a good metaDNN is heuristically oriented and it should vary from tasks to tasks. We calculate the correlation between predicted accuracies and true accuracies from the sampled architectures in evaluating the design of metaDNN. Ideally, the metaDNN is expected to rank an unseen architecture in roughly similar to its true test accuracy, i.e. corr = 1. Various ML models, such as Gaussian Process, Neural Networks, or Decision Tree, are candidates for this regression task. We choose Neural Networks as the backbone model for its powerful generalization on the high-dimensional data and the online training capability. More ablations studies for the specific choices of metaDNN is available in sec.4.2.

3.4. Transfer Learning

As MCTS incrementally builds a network with primitive actions, networks falling on the same search path render similar structures. This motivates us to incorporate Transfer Learning in AlphaX to speedup network evaluations. In simulation (Fig. 2), AlphaX recursively traverses up the tree to find a previously trained network with the minimal edit distance to the newly sampled network. Then we transfer

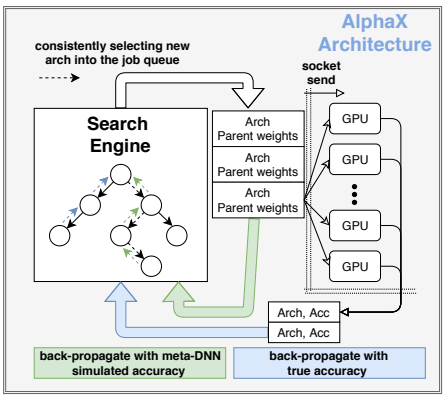


Figure 5: **Distributed AlphaX**: we decouple the original back-propagation into two parts: one uses predicted accuracy (green arrow), while the other uses the true accuracy (blue arrow). The pseudocode for the whole system is available in Appendix Sec.6

the weights of overlapping layers, and randomly initialize new layers. In the pre-training, we train every samples for 70 epochs, while we train an architecture for 20 epochs if the parent weights are available. We provide an ablation study in Fig. 12 to justify the design.

3.5. Distributed AlphaX

It is imperative to parallelize AlphaX to work on a large scale distributed systems to tackle the computation challenges rendered by NAS. Fig.5 demonstrates the distributed AlphaX. There is a master node exclusively for scheduling the search, while there are multiple clients (GPU) exclusively for training networks. The general procedures on the server side are as follows: 1) The agent follows the selection and expansion steps described in Fig.2. 2) The simulation in MCTS picks a network $arch_n$ for the actual training, and the agent traverses back to find the weights of parent architecture having the minimal edit distance to $arch_n$ for transfer learning; then we push both $arch_n$ and parent weights into a job queue. We define $arch_n$ as the selected network architecture at iteration n, and $rollout\_from(arch_n)$ as the node which it started the rollout from to reach $arch_n$. 3) The agent *preemptively back-propagates* $\hat{q} \leftarrow \frac{1}{k}\sum_{i=1..k} Pred(sim_i(s'))$ based only on predicted accuracies from the Meta-DNN at iteration n.

$$Q(s,a) \leftarrow Q(s,a) + \hat{q}, \quad N(s,a) \leftarrow N(s,a) + 1,$$
$$s \leftarrow parent(s), \quad a \leftarrow \pi_{tree}(s). \tag{4}$$

4) The server checks the receive buffer to retrieve a finished job from clients that includes $arch_z$, acc_z. Then the agent starts the second backpropagation to propagate $q \leftarrow \frac{acc_z + \hat{q}}{2}$ (Eq. 2) from the node the rollout started

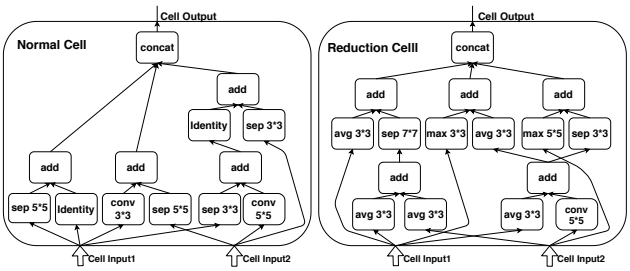
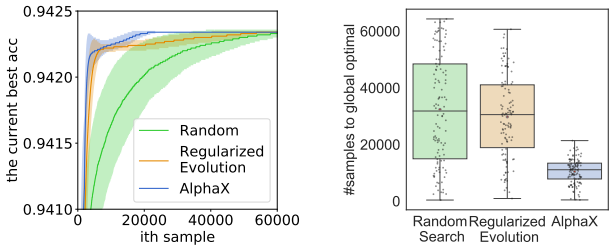


Figure 6: the $RCell$ and $NCell$ that yield the highest accuracy in the search.



(a) best acc progression (b) #samples to the best

Figure 7: **Finding the global optimum on NASBench-101**: AlphaX is 3x, 2.8x faster than Random Search and Regularized Evolution on NASBench-101 (nodes ≤ 6). The results are from 200 trails with different random seeds.

($s \leftarrow rollout\_from(arch_z)$) to replace the backpropagated $\hat{q}$ with q:

$$Q(s,a) \leftarrow Q(s,a) + q - \hat{q},$$
$$s \leftarrow parent(s), \quad a \leftarrow \pi_{tree}(s). \tag{5}$$

The client constantly tries to retrieve a job from the master job queue if it is free. It starts training once it gets the job, then it transmits the finished job back to the server. So, each client is a dedicated trainer. We also consider the fault-tolerance by taking a snapshot of the server's states every few iterations, and AlphaX can resume the searching from the breakpoint using the latest snapshot.

4. Experiment

4.1. Evaluations of Architecture Search

Experiment setup: An anonymized implementation of AlphaX to search on NASBench-101 for the reviewers are at [1]. Appendix Sec.9 provides the details of experiment setup.

4.1.1 Finding the Global Optimal on NASBench-101

Searching NASNet requires training thousands of networks, so it is not computationally feasible to conduct many trials to fairly evaluate a search algorithm. Current literatures mainly evaluate NAS algorithms with the final test accuracy, while [32] has shown many state-of-the-art NAS algorithms, e.g. DARTS [21], NAO[24], ENAS [29], cannot even outperform Random Search in the same setting. To truly evaluate a search algorithm, and to bypass the computational challenges, Christ et al collected NASBench [43] that enumerates all the possible DAGs of nodes ≤ 7, constituting of (420k+) networks and their final test accuracies.

In our experiments, we limit the maximal nodes in a DAG ≤ 6 for repeating each algorithm for 200 trails, and the rest follows the same NASBench setting, i.e. taking a subset of NASBench-101 with 64521 valid networks. The search target is the network with the highest mean test accuracy (global optimal) at 108th epochs, which can be known ahead by querying the dataset. Our evaluation metric is the number of samples to reach the best performance architecture in testing. We choose Random Search (RS) [32] and Regularized Evolution (RE) [30] as the baseline, and experimental results are in Fig.7. We have run each algorithm for 200 independent trials, and each trail is a new search fed with a different random seed. The search terminates once it reaches the target (the global optimal). Fig.7 demonstrates AlphaX is 2.8x and 3x faster than RS and RE, respectively. As we analyzed in Fig.1, Random Search lacks an online model. Regularized Evolution only mutates on top-k performing models, while MCTS explicitly builds a search tree to dynamically trade off the exploration and exploitation at individual states. Please note that the slight difference in Fig.7a actually reflects a huge gap in speed as indicated by Fig.7b. This is due to the minor difference (within 0.5%) between the near optimal architectures and the global optimal.

4.1.2 Open domain search [45]

we perform the search on CIFAR-10 using 10 NVIDIA 1080 TI. One GPU works as a server, while the rest work as clients. The client-server communications are through python sockets. To sample more architectures within limited resources, we early terminate the training at the 70th on each search clients. Then we rank architectures by their preliminary accuracies to perform an additional 530 epochs on good candidates. In acquiring the final accuracy, cutout is applied [23] using 1 random crop of size 16×16. For the subsequent ImageNet training, we construct the network for ImageNet with searched $RCell$ and $NCell$ according to Fig.2 in [45]. We set up the ImageNet training using the standard mobile configuration with the input image size of (224×224)[45]. More details are available in the appendix. AlphaX sampled 1000 networks, and we selected the top 20 networks in the pre-training to fine-tune another 530 epochs. Fig.6 demonstrates the architecture that yields the highest accuracy after fine-tuning.

Model	Space	Params	Err	GPU days	M
NASNet-A+cutout [45]	NASNet	3.3M	2.65	2000	20000
AmoebaNet-B+cutout [30]	NASNet	2.8M	$2.50_{\pm 0.05}$	3150	27000
DARTS+cutout [21]	NASNet	3.3M	$2.76_{\pm 0.09}$	4	-
RENASNet+cutout [6]	NASNet	3.5M	$2.88_{\pm 0.02}$	6	4500
AlphaX+cutout (32 filters)	NASNet	2.83M	$2.54_{\pm 0.06}$	15	1000
PNAS [17]	NASNet	3.2M	$3.41_{\pm 0.09}$	225	1160
ENAS [29]	NASNet	4.6M	3.54	0.45	-
NAONet [24]	NASNet	10.6M	3.18	200	1000
AlphaX (32 filters)	NASNet	2.83M	$3.04_{\pm 0.03}$	15	1000
NAS v3[44]	NAS	7.1M	4.47	22400	12800
Hier-EA [20]	Hier-EA	15.7M	$3.75_{\pm 0.12}$	300	7000
Proxyless-G [5]	ConvNet	5.7M.	2.08	8.33	N/A
AlphaX+cutout (64 filters)	NASNet	9.36M	$2.06_{\pm 0.04}$	15	1000

Table 2: The comparisons of our NASNet search results to other state-of-the-art results on CIFAR-10. M is the number of sampled architectures in the search. The cell structure of AlphaX is in Fig. 6.

model	multi-adds	params	latency	top1/top5 err
NASNet-A [45]	564M	5.3M	39.68ms	26.0/8.4
AmoebaNet-B [30]	555M	5.3M	32.15ms	26.0/8.5
DARTS [21]	574M	4.7M	39.32ms	26.7/8.7
RENASNet [6]	574M	4.7M	41.21ms	24.3/7.4
PNAS [17]	588M	5.1M	40.34ms	25.8/8.1
Proxyless-G [5]	-	-	83ms	25.4/7.8
AlphaX-1	579M	5.4M	38.56ms	24.5/7.8

Table 3: The error rate (%) comparisons of our best-performing architecture to other state-of-the-art results on ImageNet. The network setup follows the mobile setting defined in [45].

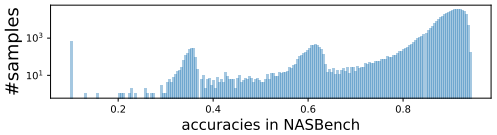


Figure 8: Accuracy distribution of networks in NASBench.

Comparisons to State-of-the-Art Results: Table. 2 and Table. 3 summarize state-of-the-art results on CIFAR10 and ImageNet, and AlphaX consistently exceeds state-of-the-art methods in both samples (M) and accuracies. Our end-to-end search cost, i.e. GPU days, is on par with SToA methods thanks to the early terminating and transfer learning. Notably, AlphaX achieves the similar accuracy to AmoebaNet with 27x less samples for the case with cutout and filters = 32. Without cutout and filters = 32, AlphaX outperforms NAONet by 0.14% in the test error with 13.3x less GPU days. Proxyless-G used a different search space, and

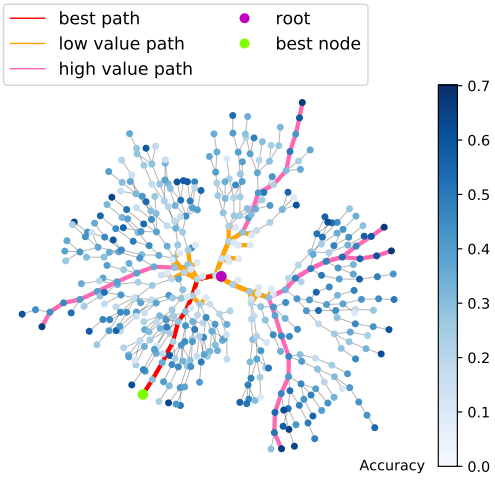
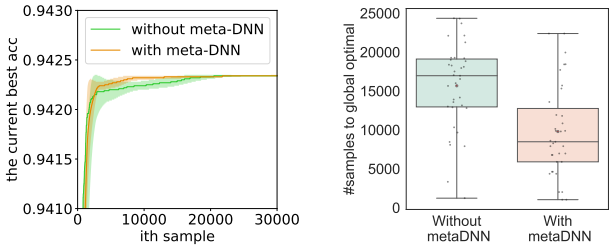


Figure 9: **AlphaX search visualization**:each nodes represents a MCTS state; the node color reflects its value, i.e. accuracy, indicating how promising a search branch.



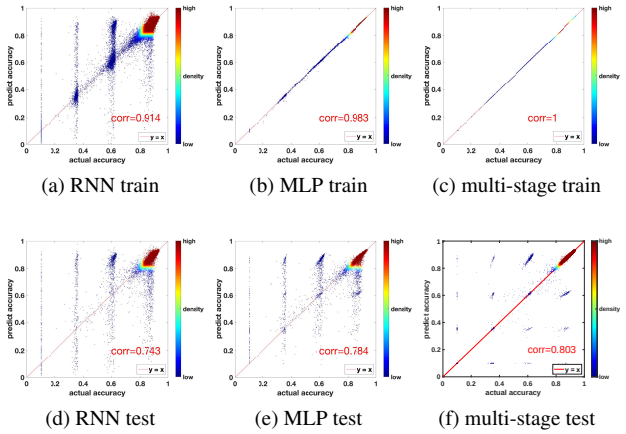
(a) best acc progression (b) #samples to the best

Figure 10: **Meta-DNN accelerates the search**: the performance of AlphaX in cases of with/without meta-DNN on NASBench-101. The data is collected from 40 trails.

our result is on par with it after increasing the filters to 64.

4.1.3 Qualitative Evaluations of AlphaX

Several interesting insights are observable in Fig.9. 1) MCTS explicitly builds a search tree to better inform future decisions based on prior rollouts. The choice of actions in MCTS is fine-grained, leveraging both the visiting statistics and value at individual states; while the random or evolutionary search is stateless, utilizing a coarse estimation of search direction (e.g. mutation). Therefore, MCTS is a lot faster than RS and RE in Fig.7. 2) MCTS invests more on the promising directions (high value path), and less otherwise (low value path). Unlike greedy based algorithms, e.g. hill climbing, MCTS consistently explores the search space as the search tree is similar to a balanced tree in Fig.9. All of



(a) RNN train (b) MLP train (c) multi-stage train

(d) RNN test (e) MLP test (f) multi-stage test

Figure 11: **meta-DNN design ablations**: True v.s. predicted accuracies of MLP, RNN and multi-stage MLP on architectures from NASBench. The scatter density is highlighted by color to reflect the data distribution; Red means high density, and blue otherwise.

these manifest that MCTS automatically balances the exploration and exploitation in the progress of the search. 3) the best performing network is not necessarily located on the most promising branch, and there are many local optimal branches (dark blue nodes). MCTS is theoretically guaranteed to find the global optimal in this non-convex search space, while PNAS or DARTs can easily trap into a local optimal.

4.2. Meta-DNN Design and its Impact

The metric in evaluating metaDNN is the correlation between the predicted v.s. true accuracy. We used 80% NASBench for training, and 20% for testing. Since DNNs have shown great success in modeling complex data, we start with Multilayer Perceptron (MLP) and Recurrent Neural Network (RNN) on building the regression model. Specific architecture details are available in appendix.10. Fig. 11d and Fig .11e demonstrate the performance of MLP (corr=0.784) is 4% better than RNN (corr=0.743), as the MLP (Fig. 11b) performs much better than RNN (Fig. 11a) in the training set. However, MLP still mispredicts many networks around 0.1, 0.4 and 0.6 and 0.8 (x-axis) as shown in Fig. 11e. This clustering effect is consistent with the architecture distribution in Fig. 8 for having many networks around these accuracies. To alleviate this issue, we propose a mult-stage model, the core idea of which is to have several dedicated MLPs to predict different ranges of accuracies, e.g. [0, 25%], along with another MLP to predict which MLP to use in predicting the final accuracy. Fig. 11f shows multi-stage model successfully improves the correlation by 1.2% from MLP, and the mispredictions have been greatly reduced. Since the multi-stage model has achieved corr =

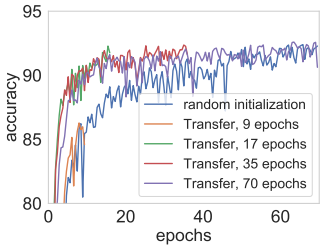
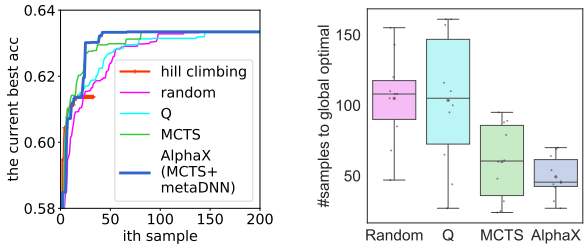


Figure 12: **Validation of transfer learning**: transferring weights significantly reduces the number of epochs in reaching the same accuracy of random initializations (Transfer 17 $\rightarrow$ 70 epochs v.s. random initialization), but insufficient epochs loses accuracy (Transfer, 9 epochs).



(a) best acc progression (b) #samples to the best

Figure 13: **Algorithmic comparisions**: AlphaX is consistently the fastest algorithm to reach the global optimal on another simplified search domain (appexdix.8), while Hill Climbing can easily trap into a local optimal.

1 on the training set, we choose it as the backbone regression model for AlphaX. Fig. 7 demonstrates our meta-DNN successfully improves the search efficiency.

4.3. Transfer Learning

The transfer learning significantly speed network evaluations up, and Fig. 12 empirically validates the effectiveness of transfer learning. We randomly sampled an architecture as the parent network. On the parent network, we added a block with two new 5x5 separable conv layers on the left and right branch as the child network. We trained the parent network toward 70 epochs and saved its weights. In training the child network, we used weights from the parent network in initializing the child network except for two new conv layers that are randomly initialized. Fig. 12 shows the accuracy progress of transferred child network at different training epochs. The transferred child network retains the same accuracy as training from scratch (random initialization) with much less epochs, but insufficient epochs loses the accuracy. Therefore, we chose 20 epochs in pre-training an architecture if transfer learning applied.

4.4. Algorithm Comparisons

Fig. 13 evaluates MCTS against Q-Learning (QL), Hill Climbing (HC) and Random Search (RS) on a simplified search space. Setup details and the introduction of the design domain are available in appendix.8. These algorithms are widely used in NAS [3, 18, 7, 10]. We conduct 10 trials for each algorithm. Fig. 13b demonstrates AlphaX is 2.3x faster than QL and RS. Though HC is the fastest, Fig. 13a indicates HC traps into a local optimal. Interestingly, Fig. 13b indicates the inter-quartile range of QL is longer than RS. This is because QL quickly converges to a suboptimal, spending a huge time to escape. This is consistent with Fig. 13a that QL converges faster than RS before the 50th samples, but random can easily escape from the local optimal afterward. Fig. 13b (MCTS v.s. AlphaX) further corroborates the effectiveness of meta-DNN.

(a) content (b) style (c) VGG (d) AlphaX-1

Figure 14: **Neural Style Transfer**: AlphaX-1 v.s. VGG.

4.5. Improved Features for Applications

CNN is a common component for Computer Vision (CV) models. Here, we demonstrate the searched architecture can improve a variety of downstream Computer Vision (CV) applications. Please check the Appendix Sec.11 for the experiment setup.

1) *Object Detection*: We replace MobileNet-v1 with AlphaX-1 in SSD [22] object detection model, and the mAP (mini-val) increases from 20.1% to 23.7% at the 300×300 resolution. (Fig.18)

2) *Neural Style Transfer*: AlphaX-1 is better than a shallow network (VGG) in capturing the rich details and textures of a sophisticated style image (Fig.19).

3) *Image Captioning*: we replace the VGG with AlphaX-1 in *show attend and tell* [42]. On the 2014 MSCOCO-val dataset, AlphaX-1 outperforms VGG by 2.4 (RELU-2), 4.4 (RELU-3), 3.7 (RELU-4), respectively (Fig.20).

5. Conclusion

We present AlphaX, the first scalable MCTS based design agent for NAS. AlphaX demonstrates superior search efficiency over mainstream algorithms on 3 different search domains, highlighting MCTS as a very promising search algorithm for NAS.

References

[1] Alphax for searching nasbench-101. https://drive.google.com/open?id=

1DrqShLtuMwFPmLsR1YuXwVzvclnuwdmz. 5

[2] P. Auer, N. Cesa-Bianchi, and P. Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine learning*, 47(2-3):235–256, 2002. 1, 3

[3] B. Baker, O. Gupta, N. Naik, and R. Raskar. Designing Neural Network Architectures using Reinforcement Learning. pages 1–18, 2016. 2, 8

[4] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games*, 4(1):1–43, 2012. 1

[5] H. Cai, L. Zhu, and S. Han. Proxylessnas: Direct neural architecture search on target task and hardware. *arXiv preprint arXiv:1812.00332*, 2018. 6

[6] Y. Chen, G. Meng, Q. Zhang, S. Xiang, C. Huang, L. Mu, and X. Wang. Renas: Reinforced evolutionary neural architecture search. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4787–4796, 2019. 6

[7] T. Elsken, J.-H. Metzen, and F. Hutter. Simple and efficient architecture search for convolutional neural networks. *arXiv preprint arXiv:1711.04528*, 2017. 2, 8

[8] C. Fernando, D. Banarse, M. Reynolds, F. Besse, D. Pfau, M. Jaderberg, M. Lanctot, and D. Wierstra. Convolution by evolution: Differentiable pattern producing networks. In *Proceedings of the Genetic and Evolutionary Computation Conference 2016*, pages 109–116. ACM, 2016. 2

[9] L. A. Gatys, A. S. Ecker, and M. Bethge. A neural algorithm of artistic style. *CoRR*, abs/1508.06576, 2015. 13

[10] K. Greff, R. K. Srivastava, J. Koutník, B. R. Steunebrink, and J. Schmidhuber. Lstm: A search space odyssey. *IEEE transactions on neural networks and learning systems*, 28(10):2222–2232, 2017. 8

[11] 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*, pages 507–523. Springer, 2011. 2

[12] R. Jozefowicz, W. Zaremba, and I. Sutskever. An empirical exploration of recurrent network architectures. In *International Conference on Machine Learning*, pages 2342–2350, 2015. 2

[13] K. Kandasamy, J. Schneider, and B. Póczos. High dimensional bayesian optimisation and bandits via additive models. In *International Conference on Machine Learning*, pages 295–304, 2015. 2

[14] L. Kocsis and C. Szepesvári. Bandit based monte-carlo planning. In *European conference on machine learning*, pages 282–293. Springer, 2006. 1, 2

[15] T. Lin, M. Maire, S. J. Belongie, L. D. Bourdev, R. B. Girshick, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick. Microsoft COCO: common objects in context. *CoRR*, abs/1405.0312, 2014. 13

[16] T. Lin, M. Maire, S. J. Belongie, L. D. Bourdev, R. B. Girshick, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick. Microsoft COCO: common objects in context. *CoRR*, abs/1405.0312, 2014. 13

[17] C. Liu, B. Zoph, J. Shlens, W. Hua, L.-J. Li, L. Fei-Fei, A. Yuille, J. Huang, and K. Murphy. Progressive neural architecture search. *arXiv preprint arXiv:1712.00559*, 2017. 2, 6, 12

[18] C. Liu, B. Zoph, J. Shlens, W. Hua, L.-J. Li, L. Fei-Fei, A. Yuille, J. Huang, and K. Murphy. Progressive Neural Architecture Search. 2017. 8

[19] H. Liu, K. Simonyan, O. Vinyals, C. Fernando, and K. Kavukcuoglu. Hierarchical Representations for Efficient Architecture Search. pages 1–13, 2017. 2

[20] H. Liu, K. Simonyan, O. Vinyals, C. Fernando, and K. Kavukcuoglu. Hierarchical representations for efficient architecture search. *arXiv preprint arXiv:1711.00436*, 2017. 6

[21] H. Liu, K. Simonyan, and Y. Yang. Darts: Differentiable architecture search. *arXiv preprint arXiv:1806.09055*, 2018. 6

[22] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. C. Berg. Ssd: Single shot multibox detector. In *European conference on computer vision*, pages 21–37. Springer, 2016. 8, 13

[23] I. Loshchilov and F. Hutter. Sgdr: Stochastic gradient descent with warm restarts. *arXiv preprint arXiv:1608.03983*, 2017. 6, 12

[24] R. Luo, F. Tian, T. Qin, E. Chen, and T.-Y. Liu. Neural architecture optimization. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems 31*, pages 7816–7827. Curran Associates, Inc., 2018. 6

[25] R. Miikkulainen, J. Liang, E. Meyerson, A. Rawal, D. Fink, O. Francon, B. Raju, H. Shahrzad, A. Navruzyan, N. Duffy, et al. Evolving deep neural networks. *arXiv preprint arXiv:1703.00548*, 2017. 2

[26] G. F. Miller, P. M. Todd, and S. U. Hegde. Designing neural networks using genetic algorithms. In *ICGA*, volume 89, pages 379–384, 1989. 2

[27] O. Nachum, M. Norouzi, and D. Schuurmans. Improving policy gradient by exploring under-appreciated rewards. *arXiv preprint arXiv:1611.09321*, 2016. 2

[28] R. Negrinho and G. Gordon. Deeparchitect: Automatically designing and training deep architectures. *arXiv preprint arXiv:1704.08792*, 2017. 2

[29] H. Pham, M. Y. Guan, B. Zoph, Q. V. Le, and J. Dean. Efficient neural architecture search via parameter sharing. *arXiv preprint arXiv:1802.03268*, 2018. 6

[30] E. Real, A. Aggarwal, Y. Huang, and Q. V. Le. Regularized evolution for image classifier architecture search. *arXiv preprint arXiv:1802.01548*, 2018. 2, 6, 12

[31] E. Real, S. Moore, A. Selle, S. Saxena, Y. L. Suematsu, J. Tan, Q. Le, and A. Kurakin. Large-Scale Evolution of Image Classifiers. 2017. 2

[32] 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. 2, 6

[33] 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*, 529(7587):484–489, 2016. 1

[34] J. Snoek, H. Larochelle, and R. P. Adams. Practical bayesian optimization of machine learning algorithms. In *Advances in neural information processing systems*, pages 2951–2959, 2012. 2

[35] K. O. Stanley. Compositional pattern producing networks: A novel abstraction of development. *Genetic programming and evolvable machines*, 8(2):131–162, 2007. 2

[36] K. O. Stanley and R. Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary computation*, 10(2):99–127, 2002. 2

[37] M. Suganuma, S. Shirakawa, and T. Nagao. A Genetic Programming Approach to Designing Convolutional Neural Network Architectures. 2017. 2

[38] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna. Rethinking the inception architecture for computer vision. *CoRR*, abs/1512.00567, 2015. 12

[39] C. Thornton, F. Hutter, H. H. Hoos, and K. Leyton-Brown. Auto-weka: Combined selection and hyperparameter optimization of classification algorithms. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 847–855. ACM, 2013. 2

[40] M. Wistuba. Finding competitive network architectures within a day using uct. *arXiv preprint arXiv:1712.07420*, 2017. 2

[41] L. Xie and A. Yuille. Genetic cnn. *arXiv preprint arXiv:1703.01513*, 2017. 2

[42] K. Xu, J. Ba, R. Kiros, K. Cho, A. Courville, R. Salakhudinov, R. Zemel, and Y. Bengio. Show, attend and tell: Neural image caption generation with visual attention. In *International conference on machine learning*, pages 2048–2057, 2015. 8

[43] 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. 1, 3, 6

[44] B. Zoph and Q. V. Le. Neural Architecture Search with Reinforcement Learning. pages 1–16, 2016. 2, 6

[45] B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le. Learning transferable architectures for scalable image recognition. *arXiv preprint arXiv:1707.07012*, 2017. 3, 6, 12

6. Pseudocode for AlphaX

In this section, we describe the pseudocode of the Distributed AlphaX. Algorithm 3 describes the search engine of AlphaX. Algorithm 2 is the server procedure to send the architecture to train chosen by the MCTS to the client and collect the architectures trained and their scores. Algorithm 1 is the client which trains and tests the architecture provided by the server.

Algorithm 1 Client

Require: Start working once building connection to the server
while True **do**
  if The client is connected to server **then**
    $network \leftarrow$ Receive()
    $accuracy \leftarrow$ Train($network$)
    Send $(network, accuracy)$ to the Server
  else
    Wait for re-connection
  end if
end while

Algorithm 2 Server

while $size(TASK\_QUEUE) > 2$ **do**
  while no idle client **do**
    Continue ▷ *Wait for dispatching jobs until there are idle clients*
  end while
  Create a new connection to a random idle client
  $network \leftarrow TASK\_QUEUE$.pop()
  Send $network$ to a Client
  if Received_Signal() **then**
    $network, accuracy \leftarrow$ Receive_Result()
    $acc(network) \leftarrow accuracy$
    $state \leftarrow rollout\_from(network)$
    Backpropagation($state$, $(accuracy - \hat{q}(state))/2$, 0)
    ▷ *Replace $\hat{q}$ with $q = Q(s, a)$ in Eq. 2*
    Train the meta-DNN with a new data $(network, accuracy)$
  else
    Continue
  end if
end while

Algorithm 3 Search Engine (MCTS)

function Expansion($state$)
  Create a new node in a tree for state.
  for all $action$ available at $state$ **do**
  $Q(state, action) \leftarrow 0, \quad N(state, action) \leftarrow 0$
  end for
end function

function Simulation($state$)
$action \leftarrow none$
  while $action$ *is not term* **do**
    randomly generate an $action$
    $next\_net \leftarrow$ Apply($state, action$)

▷ *Apply returns the next state when action is applied to state*

  end while
end function

function Backpropagation($state$, q, n)
  while $state$ *is not root* **do**
    $state \leftarrow$ parent($state$)
    $Q(state, action) \leftarrow Q(state, action) + q$
    $N(state, action) \leftarrow N(state, action) + n$
  end while
end function

Require: Start from the root
while $episode < MAX\_episode$ **do**
  Server()
  $cur\_state \leftarrow root\_node$
  $i \leftarrow 0$
  while $i < MAX\_tree\_depth$ **do**
    $i \leftarrow i + 1$
    $next\_action \leftarrow$ Selection(cur_state) ▷ *Select an action based on Eq. 1*
    if $next\_state$ not in tree **then**
      $next\_state \leftarrow$ Expansion($next\_action$)
      $T_t \leftarrow$ Simulation$_t$($next\_state$) for $t = 0...k$

▷ *k is the number of simulations we run using the Meta-DNN*

      $TASK\_QUEUE$.push(T_0)
      $rollout\_from(T_0) \leftarrow next\_state$
      $\hat{q}(next\_state) \leftarrow \frac{1}{k} \sum_{i=1..k} Pred(T_i)$

▷ *$Pred$ returns an accuracy predicted by the Meta-DNN*

      Backpropagation($next\_state, \hat{q}$) ▷ *Preemptive backpropagation to send $\hat{q}$*
    end if
  end while
end while

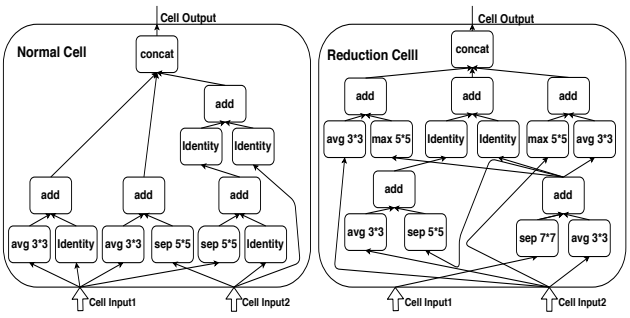


Figure 15: the $RCell$ and $NCell$ of AlphaX-2

7. Details of the State and Action Space

This section contains the description of the state and action space for NASNet design space.

We constrain the state space to make the design problem manageable. The state space exponentially grows with the depth: a k layers linear network has n^k architecture variations, where n is the number of layer types. We leverage the GPU DRAM size, our current computing resources and the design heuristics from leading DNNs, to propose the following constraints on the state space: 1) a branch has at most 1 layer; 2) a cell has at most 5 blocks; 3) the depth of blocks is limited to 2; 5) we use the layers in TABLE.4:

Actions also preserve the constraints imposed on the state space. If the next state reaches out of the design boundary, the agent automatically removes the action from the action set. For example, we exclude the "adding a new layer" action for a branch if it already has 1 layer. So, the action set is dynamically changing w.r.t states.

8. Experimental Setup for Section 4.4

The state space we consider consists of small simple CNNs. We consider a convolution layer and a softmax layer. For a convolutional layer, we allow a range of 1 or 2 for stride, 32 or 64 for filters, and 2 or 4 for kernels. We set the maximum depth to 3. We constraint that the final layer is always a dense layer with output size set to the number of classes.

Actions consist of (1) add conv layer, (2) add softmax layer, (3) increment or decrement one of the parameters for a convolution layer in the current CNN. For MCTS, random, and Q-learning agents have a terminal action to terminate the episode.

MCTS with meta-DNN: We implemented MCTS search algorithm followed the procedure with 3.2. c from Eq.1 is set to 200. The design of meta-DNN is consistent with 3.3. The meta-DNN model uses SGD optimizer with 0.9 momentum rate. All parameters in fully connected layers are initialized with the random distribution. The learning

Table 4: The code of different types of layers

layers	code	layer	code	layer	code	layer	code
3x3 avg pool	1	3x3 max pool	4	3x3 conv	7	3x3 depth-separable conv	10
5x5 avg pool	2	5x5 max pool	5	5x5 conv	8	5x5 depth-separable conv	11
7x7 avg pool	3	7x7 max pool	6	identity	9	7x7 depth-separable conv	12

rate is set to 0.0001.

MCTS without meta-DNN: we also present the results without meta-DNN, the experiment setup is consistent with above but without meta-DNN assisted simulation.

Random: agent selects action uniformly at random.

Q-learning: We implemented a tabular Q-learning agent with ϵ-greedy strategy. The learning rate is set to 0.2. We set the discount factor to be 1 in order not to prioritize short-term rewards. We fix ϵ to 0.2. We initialize the Q-value with 0.5.

Hill Climbing: For a hill climbing, an agent starts from a randomly chosen initial state. It trains every architecture in the child nodes and moves to the child node of which architecture performed the best, and repeat this procedure. Unlike MCTS and Q-learning which trains a NN only when it is a terminal state, hill climbing considers every state (and its child nodes) it visits to train. As such, we do not have a terminal action for hill climbing. As we observed that the hill climbing tends to stick to a local optimum, we restart from a randomly chosen initial state if it visits the same state twice in the trajectory.

9. Experiment Setup for Searching and Training

9.1. Setup for searching networks on NASBench

Regularized Evolution: We implemented Regularized Evolution[30] search algorithm for searching the best architecture on NASBench. The population size is set to 500. The tournament size is set to 50. We only mutate the best architecture in tournament set and replace the oldest individual in the population. Once we find the best architecture on NASBench set, we terminate the search process.

AlphaX: We implemented MCTS search algorithm followed the procedure with 3.2 on NASBench. c from Eq.1 is set to 2. The simulation times k from Eq.2 is set to 10. The design of meta-DNN is consistent with 4.2. The setup for meta-DNN is consistent with 10.

9.2. Setup for searching networks on CIFAR

We use 16 NV-1080ti gpus for searching procedure. One of them is the server running the searching program and remaining 15 gpus are clients for training the searched architectures. We use dictionary data structure in Python to save the searched architectures and convert to json file to store them in the disk. All of our training procedures are implemented in MXNET framework. The setup for training CIFAR-10 during the search are as follows: 1) We early terminate the training at the 70th epoch(3 periods of cosine restart learning rate schedule[23]) due to the limited computing resources; then we rank networks to filter out top ones to perform additional 560 epochs to acquire the final accuracy. 2) Cutout is applied [45] by using 1 crop of size 16×16. 3) Our models use cosine restart learning rate schedule[23] with 3 periods, the base learning rate is 0.05 and the batch size is 144. 4) We use the momentum optimizer with momentum rate set to 0.9 and L2 weight decay. 5) We also use dropout ratio schedule in the training. The droppath ratio is set to 0.3 and dense dropout ratio is set to 0.2 for searching procedure and applied *Schedule-DropPath*[45] for the final training. 6) We use an auxiliary classifier located at 2/3 of depth of the network. The loss of the auxiliary classifier is weighted by 0.4[38]. 7) The weights of our models are initialized with Gaussian distribution subjected to 0.01 standard deviation. 8) we randomly crop 32x32 patches from upsampled images of size 40×40 and apply random horizontal flips consistent with[45].

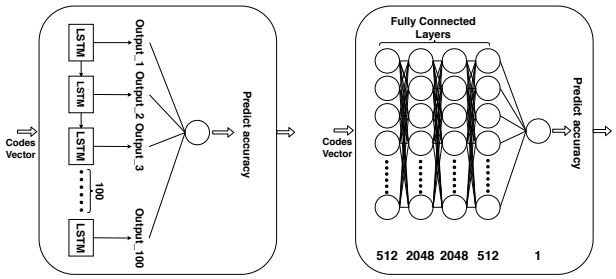
9.3. Setup for ImageNet

The setup for training ImageNet are as follows: 1) We construct the network for ImageNet with searched $RCell$ and $NCell$ according to Fig.6 in [45]. 2) The input image size is 224×224 (the mobile setting across literature). 3) Our models for ImageNet use polynomial learning rate schedule, starting with 0.05 and decay through 200 epochs. 4) We use the momentum optimizer with momentum rate set to 0.9 and L2 weight decay. 5) Our model uses an auxiliary classifier located at 2/3 of depth of the network. The loss of the auxiliary classifier is weighted by 0.4[38]. 6) Dense dropout is applied to the final softmax layer with probability 0.5. 7) We set the batch size as 256. 8) The weights of our models are initialized with Gaussian distribution subjected to 0.01 standard deviation.

10. MetaDNN Architecture and Setup

10.1. Architecture and setup for recurrent neural network (RNN)

We use an LSTM model as our predictor which is consistent with [17]. The hidden state size is set to 100. The embedding size is set to 100 as well. The final LSTM hid-



(a) RNN predictor (b) MLP predictor

Figure 16: **Architecture of metaDNN**: (a) the architecture of recurrent neural network style metaDNN (b) the architecture of multilayer perceptron style metaDNN.

den state goes through a fully-connected layer and sigmoid to regress the validation accuracy. Fig.16a shows the architecture of the RNN predictor.

The setup for training RNN predictor are as follows: 1) There are total of 20 epochs for each training. 2) The base learning late is set to 0.00002 and the batch size is 128. 3) We use the Adam optimizer for training. 4) The embeddings use uniform initialization in range [-0.1, 0.1]. the weights of fully connected layers are initialized with Uniform distribution with initial bias 0.

10.2. Architecture and setup for multilayer perceptron (MLP)

We implement both multi-stage MLP model and single MLP to predict the network accuracy. Both of multi-stage model and single MLP model share the same architecture. We use 5 fully connected layers with 512, 2048, 2048, 512 and 1 nodes. The final fully-connected layer uses the sigmoid function to regress the validation accuracy. Fig.16b shows the architecture of the MLP predictor.

The setup for training MLP predictor are as follows: 1) There are total of 20 epochs for each training. 2) The base learning late is set to 0.00002 and the batch size is 128. 3) We use the Adam optimizer for training. 4) The weights of our models are initialized with Uniform distribution with initial bias 0.

11. Setup for Vision Models

11.1. Object detection

We use AlphaX-1 model pre-trained on ImageNet dataset. The training dataset is MSCOCO for object detection[15] which contains 90 classes of objects. Each image is scaled to 300×300 in RGB channels. We trained the model with 200k iterations with 0.04 initial learning rate and the batch size is set to 24. We applied the exponential

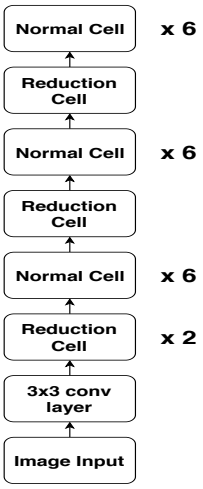


Figure 17: Cell based architecture of AlphaX model

learning rate decay schedule with the 0.95 decay factor. Our model uses momentum optimizer with momentum rate set to 0.9. We also use the L2 weight decay for training. We process each image with random horizontal flip and random crop[22]. We set the matched threshold to 0.5, which means only the probability of an object over 0.5 is effective to appear on the image. We use 8000 subsets of validation images in MSCOCO validation set and report the mean average precision (mAP) as computed with the standard COCO metric library[16].

11.2. Neural style

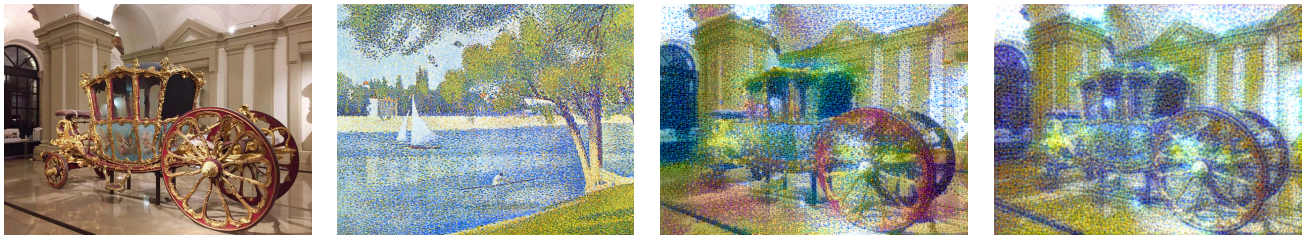
We implement the neural style transfer application by replacing the VGG model to AlplaX-1 model[9]. AlplaX-1 model is pre-trained on ImageNet dataset. In order to produce a nice result, we set the total 1000 iterations with 0.1 learning rate. We set 10 as the style weight which represents the extent of style reconstruction and 0.025 as the content weight which represents the extent of content reconstruction. We test different kinds of combinations of the outputs of different layers. Fig.17 shows the structure of AlphaX model, we found that for AlphaX-1 model, the best result can be generated by the concat layer of 13th normal cell as the feature for content reconstruction and the concat layer in first reduction cell as the feature for style reconstruction, the types of layers in each cell are shown in Fig.6 and Fig.15.

11.3. Image captioning

The training dataset of image captioning is MSCOCO[15], a large-scale dataset for the object detection, segmentation, and captioning. Each image is scaled to 224×224 in RGB channels and subtract the channel means as the input to a AlphaX-1 model. For training AlphaX-1 model, We use the SGD optimizer with the 16 batch size and the initial learning rate is 2.0. We applied

(a) AlphaX-1 (b) MobileNet-v1 (c) AlphaX-1 (d) MobileNet-v1

Figure 18: *Object Detection*: the object detection system is more precise with AlphaX-1 than MobileNet.



(a) content (b) style (c) VGG synthesized picture (d) AlphaX-1 synthesized picture

Figure 19: *Neural Style Transfer*: AlphaX-1 is better than VGG in capturing the style with sophisticated details and textures.

(a) VGG: a tennis player is playing tennis on the court.
AlphaX-1: a couple of people playing tennis on a tennis court.

(b) VGG: a bus is parked on the side of the road.
AlphaX-1: a blue bus is driving down a street.

(c) VGG: a cup of coffee and a cup of coffee.
AlphaX-1: a book and a cup of coffee on a table.

(d) VGG: a fire hydrant on the side of a road.
AlphaX-1: a red fire hydrant on the side of a road.

Figure 20: *Image Captioning*: AlphaX-1 captures more details than VGG in the captions.

the exponential learning rate decay schedule with the 0.5 decay factor in every 8 epochs.