# Neural Architecture Search Using Deep Neural Networks and Monte Carlo Tree Search

**Linnan Wang,**[*1] **Yiyang Zhao,**[*2] **Yuu Jinnai,**[1] **Yuandong Tian,**[3] **Rodrigo Fonseca**[1]
[1]Brown University [2]Worcester Polytechnic Institute [3]Facebook AI Research

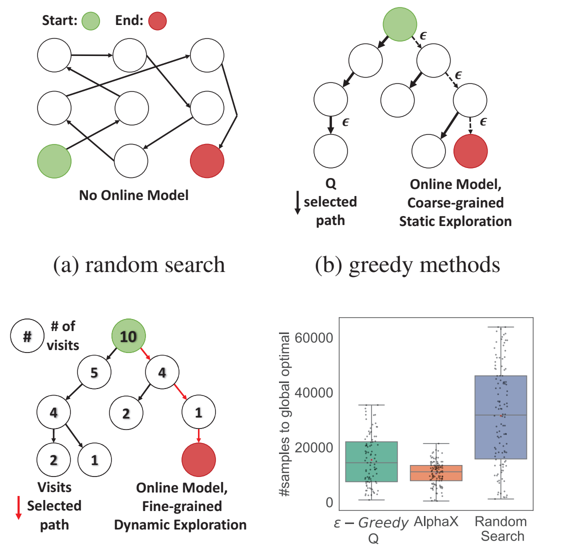
## Abstract

Neural Architecture Search (NAS) has shown great success in automating the design of neural networks, but the prohibitive amount of computations behind current NAS methods requires further investigations in improving the sample efficiency and the network evaluation cost to get better results in a shorter time. In this paper, we present a novel scalable Monte Carlo Tree Search (MCTS) based NAS agent, named AlphaX, to tackle these two aspects. AlphaX improves the search efficiency by adaptively balancing the exploration and exploitation at the state level, and by a Meta-Deep Neural Network (DNN) to predict network accuracies for biasing the search toward a promising region. To amortize the network evaluation cost, AlphaX accelerates MCTS rollouts with a distributed design and reduces the number of epochs in evaluating a network by transfer learning, which is guided with the tree structure in MCTS. In 12 GPU days and 1000 samples, AlphaX found an architecture that reaches 97.84% top-1 accuracy on CIFAR-10, and 75.5% top-1 accuracy on ImageNet, exceeding SOTA NAS methods in both the accuracy and sampling efficiency. Particularly, we also evaluate AlphaX on NASBench-101, a large scale NAS dataset; AlphaX is 3x and 2.8x more sample efficient than 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 requires many of such iterations before finding a satisfying architecture. Neural Architecture Search 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 motivate us to investigate both the search efficiency and the network evaluation cost.

AlphaGo/AlphaGoZero (Silver et al. 2016; Tian et al. 2019) have recently shown super-human performance in playing the game of Go, by using a specific search algorithm called Monte-Carlo Tree Search (MCTS) (Kocsis and Szepesvári 2006). 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. The 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 the traditional MCTS approach that evalu-




(a) random search (b) greedy methods

(c) MCTS (d) 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., $\epsilon$-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. The numbers in nodes represent values. **(d)** Performance of different search algorithms on NASBench-101. AlphaX is 3x, 1.5x more sample-efficient than *random search* and $\epsilon$-*greedy* based *Q-learning*.

[*]Linnan Wang and Yiyang Zhao contributed equally

| Methods | Global Solution | Online Model | Exploration | Distributed Ready | Transfer Learning |
|---|---|---|---|---|---|
| MetaQNN (QL) [1] | × | - | $\epsilon$-greedy | × | × |
| Zoph (PG) [2] | × | RNN | - | √ | × |
| PNAS (HC) [3] | × | RNN | - | √ | × |
| Regularized Evolution [4] | √ | - | top-k mutation | √ | × |
| Random Search [5] | √ | - | random | √ | × |
| DeepArchitect (MCTS) [6] | √ | - | UCT | × | × |
| Wistuba (MCTS) [7] | √ | Gaussian | UCT | × | × |
| AlphaX (MCTS) | √ | meta-DNN | UCT | √ | √ |

Table 1: Highlights of NAS search algorithms: compared to DeepArchitect and Wistuba, AlphaX makes several contributions toward a practical MCTS based NAS agent by improving the sample efficiency with an effective multi-stage metaDNN, and amortizes the network evaluation costs by a distirbuted design and transfer learning.

ates 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) (Auer, Cesa-Bianchi, and Fischer 2002), based 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, a NAS agent that uses MCTS for efficient 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 search trials. Fig. 1 summarizes the trade-offs. Furthermore, while prior works applied MCTS to Architecture Search (Wistuba 2017; Negrinho and Gordon 2017), they lack an effective model to accurately predict rewards, and the expensive network evaluations in MCTS rollouts still remain unaddressed. Toward a practical MCTS-based NAS agent, AlphaX has two novel features: first, a highly accurate multi-stage meta-DNN to improve the sample efficiency; and second, the use of transfer learning, together with a scalable distributed design, to amortize the network evaluation costs. As a result, AlphaX is the first MCTS-based NAS agent that reports SOTA accuracies on both CIFAR-10 and ImageNet in on par with the SOTA end-to-end search time.

[1](Baker et al. 2016)
[2](Zoph and Le 2016)
[3](Liu et al. 2017a)
[4](Real et al. 2018)
[5](Sciuto et al. 2019)
[6](Negrinho and Gordon 2017)
[7](Wistuba 2017)

## 2 Related Work

*Monte Carlo Tree Search:* DeepArchitect (Negrinho and Gordon 2017) implemented vanilla MCTS for NAS without a predictive model, and Wistuba (Wistuba 2017) uses statistics from the current search (e.g., RAVE and Contextual Reward Prediction) to predict the 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. Besides, the prior works do not address the expensive network evaluations in MCTS rollouts, while AlphaX solves this issue with a distributed design and transfer learning, making important improvements toward a practical MCTS based NAS agent.

*Bayesian Optimization* (BO) is a popular method for the hyper-parameter search (Kandasamy, Schneider, and Póczos 2015). BO has proven to be successful for small scale problems, e.g. hyper-parameter tuning of Stochastic Gradient Descent (SGD). Though BO variants, e.g. TPE (Bergstra et al. 2011), have tackled the cubic scaling issue in Gaussian Process, optimizing the acquisition function in a high dimensional search space that contains over $10^{17}$ neural architectures become the bottleneck.

*Reinforcement Learning (RL)*: Several RL techniques have been investigated for NAS (Baker et al. 2016; Zoph and Le 2016). Baker et al. proposed a Q-learning agent to design network architectures (Baker et al. 2016). The agent takes a $\epsilon$-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 (Zoph and Le 2016). However, directly maximizing the expected reward in Policy Gradient could lead to local optimal solution (Nachum, Norouzi, and Schuurmans 2016).

*Hill Climbing (HC)*: Elsken et al. proposed a simple hill climbing for NAS (Elsken, Metzen, and Hutter 2017). 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 (Liu et al. 2017a). 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 optimum given enough samples (Kocsis and Szepesvári 2006).

*Evolutionary Algorithm (EA)*: Evolutionary algorithms represent each neural network as a string of genes and search the architecture space by mutation and recombinations (Real et al. 2018). 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 it lacks an online model to inform decisions.

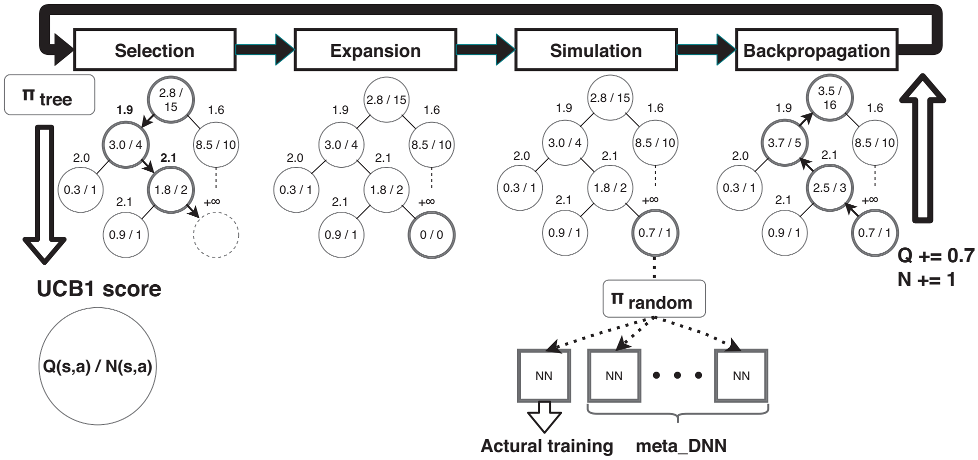
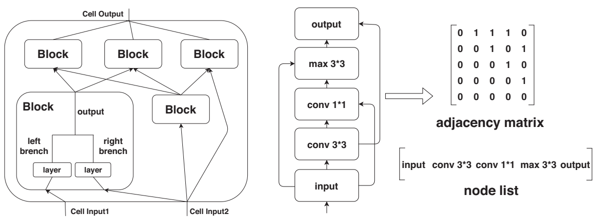



Figure 2: Overview of AlphaX search procedures: explanations of four steps are in sec.3.2.




(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. object detection and 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*: (Zoph et al. 2017) proposes searching a hierarchical Cell structure as shown in Fig.3a. There are two types of Cells, Normal Cell ($NCell$) and Reduction Cell ($RCell$). $NCell$ maintains the input and output dimensions with the padding, while $RCell$ reduces the height and width by half with the striding. Then, the network is constituted by stacking multiple cells.
- *NASBench Search Space*: (Ying et al. 2019) proposes searching a small Direct Acyclic Graph (DAG) with each node representing a layer and each edge representing the inter-layer dependency as shown in Fig.3b. 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 layer 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 in a cell, creating a new block with different input combinations, and the terminating action. The actions for the NASBench search domain are either adding a node or an edge, and the terminating action.

## 3.2 Search Procedure

This section elaborates the integration of MCTS and metaDNN. The purpose of MCTS is to analyze the most promising move at a state, while the purpose of 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 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 is. 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 (Auer, Cesa-Bianchi, and Fischer 2002):

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

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.

***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

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

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 to 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-

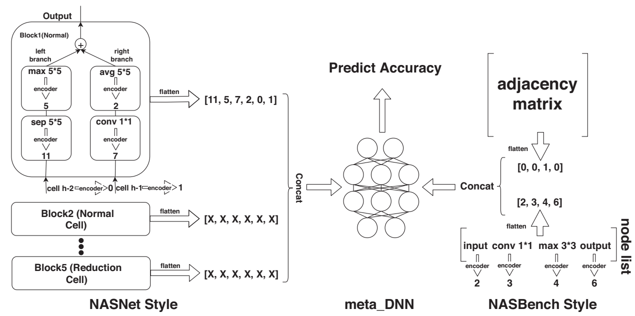



Figure 4: Encoding scheme of NASBench and NASNet.

propagate the information to its ancestral as:

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

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 $\leq 7$, $7 \times 7$ (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

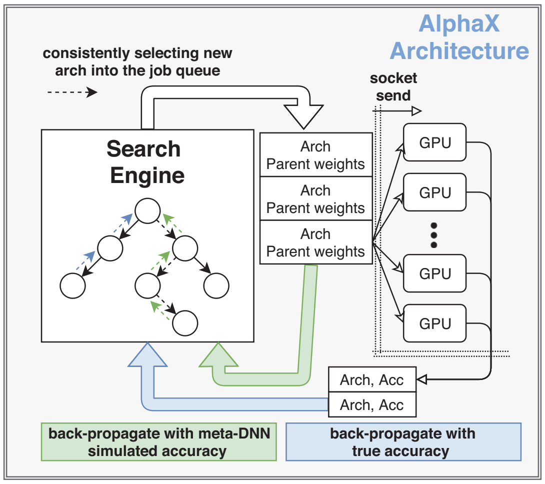



Figure 5: Distributed AlphaX: we decouple the original back-propagation into two parts: one uses predicted accuracy (green arrow), while the other uses the true accuracy (blue arrow). The pseudocode for the whole system is available in Appendix Sec.A

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 are 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 speed network evaluations up. 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 the weights of overlapping layers, and randomly initialize new layers. In the pre-training, we train every sample for 70 epochs if no parent networks are transferable, and 20 epochs otherwise. Fig. 11 provides a study to justify the design.

## 3.5 Distributed AlphaX

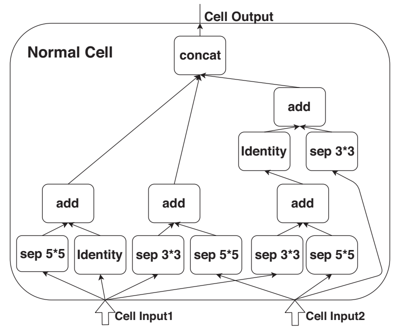
It is imperative to parallelize AlphaX to work on a large scale distributed systems to tackle the computation challenges rendered by NAS. Fig.5 demonstrates the distributed AlphaX. There is a master node exclusively for scheduling the search, while there are multiple clients (GPU) exclusively for training 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 backpropagates* $\hat{q} \leftarrow \frac{1}{k}\sum_{i=1..k} Pred(sim_i(s'))$ based only on predicted accuracies from the Meta-DNN at iteration $n$.

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

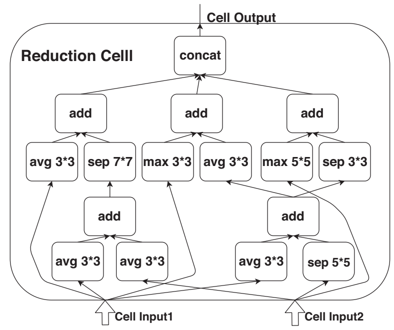
4) The server checks the receive buffer to retrieve a finished job from clients that includes $arch_z$, $acc_z$. Then the agent starts the second backpropagation to propagate $q \leftarrow \frac{acc_z + \hat{q}}{2}$ (Eq. 2) from the node the rollout started ($s \leftarrow rollout\_from(arch_z)$) to replace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




(a) normal cell




(b) reduction cell

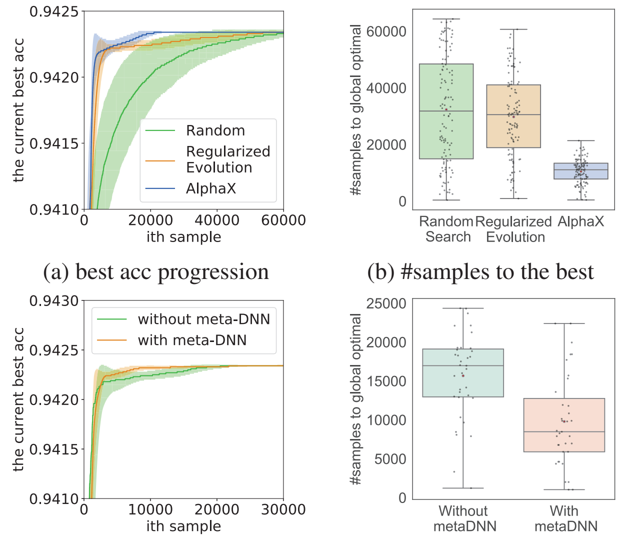
Figure 6: the normal and reduction cells that yield the highest accuracy in the search.

| Model | Params | Err | GPU days | M |
|---|---|---|---|---|
| NASNet-A+cutout (Zoph et al. 2017) | 3.3M | 2.65 | 2000 | 20000 |
| AmoebaNet-B+cutout (Real et al. 2018) | 2.8M | $2.50_{\pm 0.05}$ | 3150 | 27000 |
| DARTS+cutout (Liu et al. 2018) | 3.3M | $2.76_{\pm 0.09}$ | 4 | - |
| RENASNet+cutout (Chen et al. 2019) | 3.5M | $2.88_{\pm 0.02}$ | 6 | 4500 |
| AlphaX+cutout (32 filters) | 2.83M | $2.54_{\pm 0.06}$ | 12 | 1000 |
| PNAS (Liu et al. 2017a) | 3.2M | $3.41_{\pm 0.09}$ | 225 | 1160 |
| ENAS (Pham et al. 2018) | 4.6M | 3.54 | 0.45 | - |
| NAONet (Luo et al. 2018) | 10.6M | 3.18 | 200 | 1000 |
| AlphaX (32 filters) | 2.83M | $3.04_{\pm 0.03}$ | 12 | 1000 |
| NAS v3(Zoph and Le 2016) | 7.1M | 4.47 | 22400 | 12800 |
| Hier-EA (Liu et al. 2017c) | 15.7M | $3.75_{\pm 0.12}$ | 300 | 7000 |
| AlphaX+cutout (128 filters) | 31.36M | $2.16_{\pm 0.04}$ | 12 | 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 | top1/top5 err |
|---|---|---|---|
| NASNet-A (Zoph et al. 2017) | 564M | 5.3M | 26.0/8.4 |
| AmoebaNet-B (Real et al. 2018) | 555M | 5.3M | 26.0/8.5 |
| DARTS (Liu et al. 2018) | 574M | 4.7M | 26.7/8.7 |
| RENASNet (Chen et al. 2019) | 574M | 4.7M | 24.3/7.4 |
| PNAS (Liu et al. 2017a) | 588M | 5.1M | 25.8/8.1 |
| AlphaX-1 | 579M | 5.4M | 24.5/7.8 |

Table 3: Transferring the CIFAR architecture to ImageNet, and their results in the mobile setting.




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

(c) best acc progression (d) #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 $\leq 6$). The results are from 200 trails with different random seeds. (c) and (d) show the performance of AlphaX in cases of with/without meta-DNN on NASBench-101

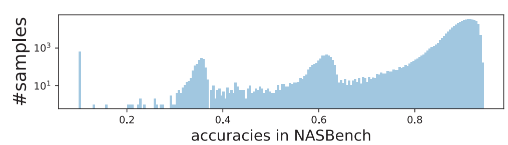



Figure 8: Accuracy distribution of NASBench.

# 4 Experiments

## 4.1 Evaluations of architecture search

***Open domain search***: we perform the search on CIFAR-10 using 8 NVIDIA 1080 TI. One GPU works as a server, while the rest work as clients. To further speedup network evaluations, we early terminated the training at 70th epoch during the pre-training. We selected the top 20 networks from the pre-training and fine-tuned them additional 530 epochs to get the final accuracy. For the ImageNet training, we constructed the network with the same $RCell$ and $NCell$ searched on CIFAR10 following the accepted standard, i.e. the mobile setting, defined in (Zoph et al. 2017). In total, AlphaX sampled 1000 networks; and Fig. 6 demonstrates the architecture that yields the highest accuracy after fine-tuning. More details are available at appendix. C.

Table. 2 and Table. 3 and summarize SOTA results on CIFAR10 and ImageNet, and AlphaX achieves SOTA accuracy with the least samples (M). Our end-to-end search cost, i.e. GPU days, is also on par with SOTA methods due to the early terminating and transfer learning. Notably, AlphaX achieves similar accuracy to AmoebaNet with 27x fewer 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 16.7x fewer GPU days.

***Searching on NAS dataset***: To further examine the sample efficiency, we evaluate AlphaX on the recent NAS dataset, NASBench-101 (Ying et al. 2019). NASBench enumerates all the possible DAGs of nodes $\leq 7$, constituting of (420k+) networks and their final test accuracies. This enables bypassing the computation barrier to fairly evaluate the sample efficiency. In our experiments, we limited the maximal nodes in a DAG $\leq 6$, i.e. constructing a subset of NASBench-101 that contains 64521 valid networks. This allows us to quickly repeat each algorithm for 200 trials. The search target is the network with the highest mean test accuracy (the global optimum) at 108th epochs, which can be known ahead by querying the dataset. We choose Random Search (RS) (Sciuto et al. 2019) and Regularized Evolution (RE) (Real et al. 2018) as the baseline, as RE delivers competitive results according to Table. 2 in the NASNet search space, and RS finds the global optimal in expected n/2, where n is the dataset size.

Fig. 7 demonstrates AlphaX is 2.8x and 3x faster than RE and RS, 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. From Fig. 8, it shows there are abundant architectures with minor performance difference to

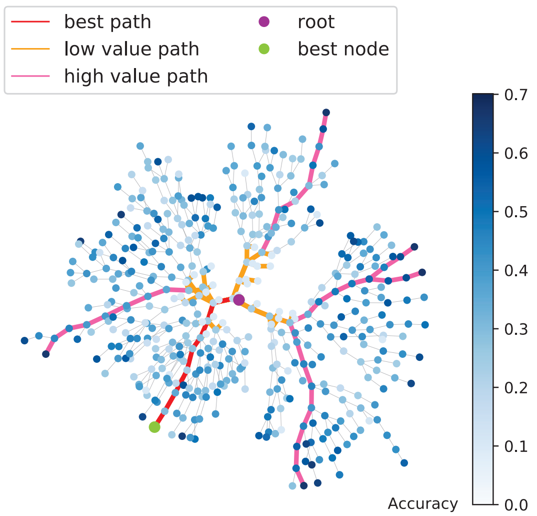
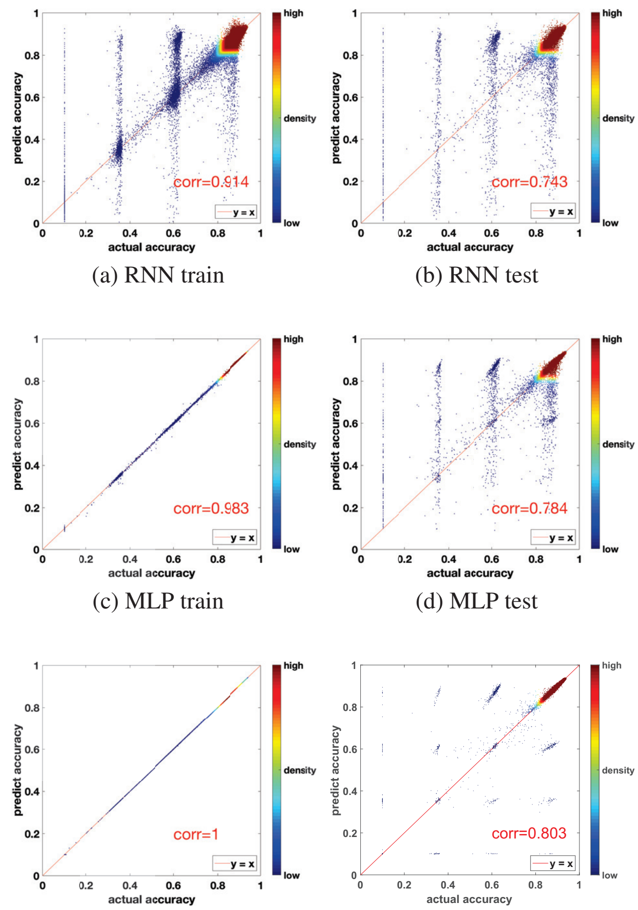



Figure 9: AlphaX search visualization:each node represents a MCTS state; the node color reflects its value, i.e. accuracy, indicating how promising a search branch.




(a) RNN train (b) RNN test

(c) MLP train (d) MLP test

(e) multi-stage train (f) multi-stage test

Figure 10: 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.

the global optimum. Therefore, it is fast to find the top 5% architectures, while slow in reaching the global optimum.

***Qualitative evaluations of AlphaX***: Several interesting insights are observable in Fig.9. 1) 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 guided by the adaptive UCT. 2) the best performing network is not necessarily located on the most promising branch, highlighting the importance of exploration in NAS.

## 4.2 Component evaluations

***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.E. Fig. 10b and Fig .10d demonstrate the performance of MLP (corr=0.784) is $4\%$ better than RNN (corr=0.743), as the MLP (Fig. 10c) performs much better than RNN (Fig. 10a) 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. 10d. 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 multi-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. 10f shows a 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 = 1 on the training set, we choose it as the backbone regression model for AlphaX. Fig. 7 demonstrates our meta-DNN is effective to sustain NAS.

***Transfer Learning***: the transfer learning significantly speeds network evaluations up, and Fig. 11 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. 11 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 fewer epochs, but insufficient epochs lose the accuracy. Therefore, we chose 20 epochs in

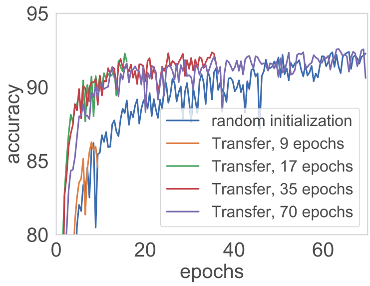
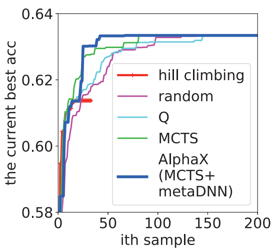
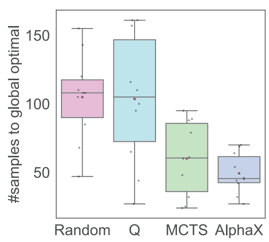



Figure 11: Validation of transfer learning: transferring weights significantly reduces the number of epochs in reaching the same accuracy of random initializations (Transfer 17 $\rightarrow$ 70 epochs v.s. random initialization), but insufficient epochs loses accuracy (Transfer, 9 epochs).




(a) best acc progression




(b) #samples to the best

Figure 12: Algorithmic comparisons: AlphaX is consistently the fastest algorithm to reach the global optimal on another simplified search domain (appendix.D), while Hill Climbing can easily trap into a local optimal.

pre-training an architecture if transfer learning applied.

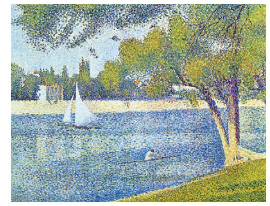
### 4.3 Algorithm Comparisons

Fig. 12 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.D. These algorithms are widely used in NAS (Baker et al. 2016; Liu et al. 2017b; Elsken, Metzen, and Hutter 2017; Greff et al. 2017). We conduct 10 trials for each algorithm. Fig. 12b demonstrates AlphaX is 2.3x faster than QL and RS. Though HC is the fastest, Fig. 12a indicates HC traps into a local optimal. Interestingly, Fig. 12b 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. 12a that QL converges faster than RS before the 50th samples, but random can easily escape from the local optimal afterward. Fig. 12b (MCTS v.s. AlphaX) further corroborates the effectiveness of meta-DNN.

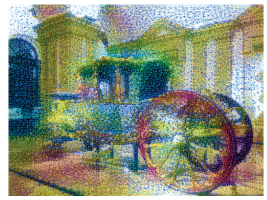
### 4.4 Improved Features for Vision Applications

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.F for the experiment setup.

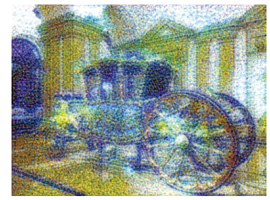
(a) content



(b) style



(c) VGG



(d) AlphaX-1

Figure 13: Neural Style Transfer: AlphaX-1 v.s. VGG.

1) *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. 13).

2) *Object Detection*: We replace MobileNet-v1 with AlphaX-1 in SSD (Liu et al. 2016) object detection model, and the mAP (mini-val) increases from $20.1\%$ to $23.7\%$ at the $300 \times 300$ resolution. (Fig.16, appendix).

3) *Image Captioning*: we replace the VGG with AlphaX-1 in *show attend and tell* (Xu et al. 2015). 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.17, appendix).

## 5 Conclusion

In this paper, we propose a new MCTS based NAS agent named AlphaX. Compared to prior MCTS agents, AlphaX is the first practical MCTS based agent that achieves SOTA results on both CIFAR-10 and ImageNet in a reasonable amount of computations by improving the sampling efficiency with a novel predictive model meta-DNN, and by amortizing the network evaluation costs with a scalable solution and transfer learning. In 3 search tasks, AlphaX consistently demonstrates superior search efficiency over mainstream algorithms, highlighting MCTS as a promising search algorithm for NAS.

## 6 Appendix

The appendix is available at
https://arxiv.org/abs/1805.07440.

## References

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

Baker, B.; Gupta, O.; Naik, N.; and Raskar, R. 2016. Designing Neural Network Architectures using Reinforcement Learning. 1–18.

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

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

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

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

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

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

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

Liu, C.; Zoph, B.; Shlens, J.; Hua, W.; Li, L.-J.; Fei-Fei, L.; Yuille, A.; Huang, J.; and Murphy, K. 2017a. Progressive neural architecture search. *arXiv preprint arXiv:1712.00559*.

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

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

Liu, H.; Simonyan, K.; Simonyan, K.; and Yang, Y. 2018. Darts: Differentiable architecture search. *arXiv preprint arXiv:1806.09055*.

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

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

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

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

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

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

Silver, D.; Huang, A.; Maddison, C. J.; Guez, A.; Sifre, L.; Van Den Driessche, G.; Schrittwieser, J.; Antonoglou, I.; Panneershelvam, V.; Lanctot, M.; et al. 2016. Mastering the game of go with deep neural networks and tree search. *nature* 529(7587):484–489.

Tian, Y.; Ma, J.; Gong, Q.; Sengupta, S.; Chen, Z.; Pinkerton, J.; and Zitnick, C. L. 2019. ELF opengo: An analysis and open reimplementation of alphazero. *CoRR* abs/1902.04522.

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

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

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

Zoph, B., and Le, Q. V. 2016. Neural Architecture Search with Reinforcement Learning. 1–16.

Zoph, B.; Vasudevan, V.; Shlens, J.; and Le, Q. V. 2017. Learning transferable architectures for scalable image recognition. *arXiv preprint arXiv:1707.07012*.