AUTO-KERAS: EFFICIENT NEURAL ARCHITECTURE SEARCH WITH NETWORK MORPHISM

Haifeng Jin [1] **Qingquan Song** [1] **Xia Hu** [1]

ABSTRACT

Neural architecture search (NAS) has been proposed to automatically tune deep neural networks, but existing search algorithms usually suffer from expensive computational cost. Network morphism, which keeps the functionality of a neural network while changing its neural architecture, could be helpful for NAS by enabling a more efficient training during the search. In this paper, we propose a novel framework enabling Bayesian optimization to guide the network morphism for efficient neural architecture search by introducing a neural network kernel and a tree-structured acquisition function optimization algorithm, which more efficiently explores the search space. Intensive experiments have been done to demonstrate the superior performance of the developed framework over the state-of-the-art methods. Moreover, we build an open-source AutoML system on our method, namely Auto-Keras[2]. The system runs in parallel on CPU and GPU, with an adaptive search strategy for different GPU memory limits.

1 INTRODUCTION

Automated Machine Learning (AutoML) has become a very important research topic with the wide application of machine learning techniques. The goal of AutoML is to enable people with limited machine learning background knowledge to use the machine learning models easily. In the context of deep learning, neural architecture search (NAS), which aims to search for the best neural network architecture for the given learning task and dataset, has become an effective computational tool in AutoML. Unfortunately, existing NAS algorithms are usually computationally expensive. The time complexity of NAS can be seen as $O(n\bar{t})$, where n is the number of neural architectures evaluated during the search, and $\bar{t}$ is the average time consumption for evaluating each of the n neural networks. Many NAS approaches, such as deep reinforcement learning (Zoph & Le, 2016; Baker et al., 2016; Zhong et al., 2017; Pham et al., 2018) and evolutionary algorithms (Real et al., 2017; Desell, 2017; Liu et al., 2017; Suganuma et al., 2017; Xie & Yuille, 2017; Real et al., 2018), require a large n to reach a good performance. Also, each of the n neural networks is trained from scratch which is very slow.

Network morphism has also been explored for neural architecture search (Cai et al., 2018; Elsken et al., 2017). Network morphism is a technique to morph the architecture of a neural network but keep its functionality (Chen et al., 2015; Wei et al., 2016). Therefore, we are able to modify a trained neural network into a new architecture using the network morphism operations, *e.g.*, inserting a layer or adding a skip-connection. Only a few more epochs are required to further train the new architecture for better performance. Using network morphism would reduce the average training time $\bar{t}$ in neural architecture search. The most important problem to solve for network morphism based NAS methods is the selection of operations, which is to select from the network morphism operation set to morph an existing architecture to a new one. The state-of-the-art network morphism based method (Cai et al., 2018) uses a deep reinforcement learning controller, which requires a large number of training examples, *i.e.*, n in $O(n\bar{t})$. Another simple approach (Elsken et al., 2017) is to use random algorithm and hill-climbing, which can only explore the neighborhoods of the searched area each time, and could potentially be trapped by local optimum. How to perform efficient neural architecture search with network morphism remains a challenging problem.

As we know, Bayesian optimization (Snoek et al., 2012) has been widely adopted to efficiently explore black-box functions for global optimization, whose observations are expensive to obtain. For example, it has been used in hyperparameter tuning for machine learning models (Thornton et al., 2013; Kotthoff et al., 2016; Feurer et al., 2015; Hutter et al., 2011), in which Bayesian optimization searches among different combinations of hyperparameters. During the search, each evaluation of a combination of hyperparameters involves an expensive process of training and testing of the machine learning model, which is very similar to the NAS problem. The unique properties of Bayesian opti-

[1] Department of Computer Science and Engineering, Texas A&M University, College Station, United States.
[2] The code is available at http://autokeras.com.

mization motivate us to explore its capability in guiding the network morphism to reduce the number of trained neural networks n to make the search more efficient.

It is a non-trivial task to design a Bayesian optimization method for network morphism based neural architecture search due to the following challenges. First, the underlying Gaussian process (GP) is traditionally used for Euclidean space. To update the Bayesian optimization model with observations, the underlying GP is to be trained with the searched architectures and their performance. However, the neural network architectures are not in Euclidean space and hard to parameterize into a fixed-length vector. Second, an acquisition function needs to be optimized for Bayesian optimization to generate the next architecture to observe. However, it is not to maximize a function in Euclidean space for morphing the neural architectures, but to select a node to expand in a tree-structured search space, where each node represents an architecture and each edge a morph operation. Thus traditional Newton-like or gradient-based methods cannot be simply applied. Third, the network morphism operations are complicated in a search space of neural architectures with skip-connections. A network morphism operation on one layer may change the shapes of some intermediate output tensors, which no longer match input shape requirements of the layers taking them as input. How to maintain such consistency is not defined in previous work on network morphism.

In this paper, an efficient neural architecture search with network morphism is proposed, which utilizes Bayesian optimization to guide through the search space by selecting the most promising operations each time. To tackle the aforementioned challenges, an edit-distance based neural network kernel is constructed. Being consistent with the key idea of network morphism, it measures how many operations are needed to change one neural network to another. Besides, a novel acquisition function optimizer, which is capable of balancing between the exploration and exploitation, is designed specially for the tree-structure search space to enable Bayesian optimization to select from the operations. In addition, a graph-level morphism is defined to address the complicated changes in the neural architectures based on previous layer-level network morphism. The proposed approach is compared with the state-of-the-art NAS methods (Kandasamy et al., 2018; Elsken et al., 2017) on benchmark datasets of MNIST, CIFAR10, and FASHION-MNIST. Within a limited search time, the architectures found by our method achieves the lowest error rates on all of the datasets.

In addition, we develop an open-source AutoML system based on our proposed method, namely Auto-Keras, which can be download and installed locally. The system is carefully designed with a concise interface for people not specialized in computer programming and data science to use. To speed up the search, the workload on CPU and GPU can run in parallel. To address the issue of different GPU memory, which limits the size of the neural architectures, a memory adaption strategy is designed for deployment.

The main contributions of the paper are as follows:

- An algorithm for efficient neural architecture search with Bayesian optimization guided network morphism is proposed.
- Propose a neural network kernel for Bayesian optimization, a tree-structured acquisition function optimizer, and a graph-level morphism.
- Conduct intensive experiments on benchmark datasets to demonstrate the superior performance of the proposed method over the baseline methods.
- An open-source AutoML system, namely Auto-Keras, is developed based on our method for neural architecture search.

2 Problem Statement

The general neural architecture search problem we studied in this paper is defined as: given a neural architecture search space $\mathcal{F}$, the input data $\boldsymbol{X}$, and the cost metric $Cost(\cdot)$, we aim at finding an optimal neural network $f^* \in \mathcal{F}$ with its trained parameter $\boldsymbol{\theta_{f^*}}$, which could achieve the lowest cost metric value on the given dataset $\boldsymbol{X}$. Mathematically, this definition is equivalent to finding f^* satisfing:

$$f^* = \underset{f \in \mathcal{F}}{\operatorname{argmin}} \min_{\boldsymbol{\theta_f}} Cost(f(\boldsymbol{X}; \boldsymbol{\theta_f})), \tag{1}$$

where $\boldsymbol{\theta_f} \in \mathbb{R}^{w(f)}$ denotes the parameter set of network f, $w(f)$ is the number of parameters in f.

Before explaining the proposed algorithm, we first define the target search space $\mathcal{F}$. Let $G_f = (V_f, E_f)$ denotes the computational graph of a neural network f. Each node $v \in V_f$ denotes an intermediate output tensor of a layer of f. Each directed edge $e_{u \to v} \in E_f$ denotes a layer of f, whose input tensor is $u \in V_f$ and output tensor is $v \in V_f$. $u \prec v$ indicates v is before u in topological order of the nodes, *i.e.*, by traveling through the edges in E_f, u is reachable from node v. The search space $\mathcal{F}$ in this work is defined as: a space consisting of any neural network architecture f, which satisfies two conditions: (1) G_f is a directed acyclic graph (DAG). (2) $\forall u\ v \in V_f$, $(u \prec v) \vee (v \prec u)$. It is worth pointing out that although skip connection is allowed, there should only be one main chain in f. Moreover, the search space $\mathcal{F}$ defined here is large enough to cover a wide range of famous neural architectures, *e.g.*, DenseNet, ResNet.

3 Network Morphism with Bayesian Optimization

The key idea of the proposed method is to explore the search space via morphing the network architectures guided by an efficient Bayesian optimization algorithm. Traditional Bayesian optimization consists of a loop of three steps: update, generation, and observation. In the context of NAS, our proposed Bayesian optimization algorithm iteratively conducts: (1) **Update**: train the underlying Gaussian process model with the existing architectures and their performance; (2) **Generation**: generate the next architecture to observe by optimizing an delicately defined acquisition function; (3) **Observation**: train the generated neural architecture to obtain the actual performance. There are three main challenges in designing the method for morphing the neural architectures with Bayesian optimization. We introduce three key components separately in the subsequent sections coping with the three design challenges.

3.1 Neural Network Kernel for Gaussian Process

The first challenge we need to address is that the NAS space is not a Euclidean space, which does not satisfy the assumption of the traditional Gaussian process. Directly vectorizing the neural architecture is impractical due to the uncertain number of layers and parameters it may contain. Since the Gaussian process is a kernel method, instead of vectorizing a neural architecture, we propose to tackle the challenge by designing a neural network kernel function. The intuition behind the kernel function is the edit-distance for morphing one neural architecture to another.

Kernel Definition: Suppose f_a and f_b are two neural networks. Inspired by Deep Graph Kernels (Yanardag & Vishwanathan, 2015), we propose an edit-distance kernel for neural networks, which accords with our idea of employing network morphism. Edit-distance here means how many operations are needed to morph one neural network to another. The concrete kernel function is defined as follows:

$$\kappa(f_a, f_b) = e^{-\rho^2(d(f_a,f_b))}, \tag{2}$$

where function $d(\cdot,\cdot)$ denotes the edit-distance of two neural networks, whose range is $[0, +\infty)$, ρ is a mapping function, which maps the distance in the original metric space to the corresponding distance in the new space. The new space is constructed by embedding the original metric space into a new one using Bourgain Theorem (Bourgain, 1985), the purpose of which is to ensure the validity of the kernel. More details are introduced in Theorem 2.

Calculating the edit-distance of two neural networks can be mapped to calculating the edit-distance of two graphs, which is an NP-hard problem (Zeng et al., 2009). Based on the search space $\mathcal{F}$ we have defined in Section 2, we solve the problem by proposing an approximated solution as follows:

$$d(f_a, f_b) = D_l(L_a, L_b) + \lambda D_s(S_a, S_b), \tag{3}$$

where D_l denotes the edit-distance for morphing the layers, *i.e.*, the minimum edit needed to morph f_a to f_b if the skip-connections are ignored, $L_a = \{l_a^{(1)}, l_a^{(2)}, \dots\}$ and $L_b = \{l_b^{(1)}, l_b^{(2)}, \dots\}$ are the layer sets of neural networks f_a and f_b, D_s is the approximated edit-distance for morphing skip-connections between two neural networks, $S_a = \{s_a^{(1)}, s_a^{(2)}, \dots\}$ and $S_b = \{s_b^{(1)}, s_b^{(2)}, \dots\}$ are the skip-connection sets of neural network f_a and f_b, and λ is the balancing factor.

Calculating D_l: We assume $|L_a| < |L_b|$, the edit-distance for morphing the layers of two neural architectures f_a and f_b is calculated by minimizing the follow equation:

$$D_l(L_a, L_b) = \min \sum_{i=1}^{|L_a|} d_l(l_a^{(i)}, \varphi_l(l_a^{(i)})) + \Big| |L_b| - |L_a| \Big|, \tag{4}$$

where $\varphi_l : L_a \to L_b$ is an injective matching function of layers satisfying: $\forall i < j, \varphi_l(l_a^{(i)}) \prec \varphi_l(l_a^{(j)})$ if layers in L_a and L_b are all sorted in topological order. $d_l(\cdot,\cdot)$ denotes the edit-distance of widening a layer into another defined in Equation (5), where $w(l)$ is the width of layer l.

$$d_l(l_a, l_b) = \frac{|w(l_a) - w(l_b)|}{max[w(l_a), w(l_b)]}. \tag{5}$$

The intuition of Equation (4) is consistent with the idea of network morphism shown in Figure 1. Suppose a matching is provided between the nodes in two neural networks. The size of the tensors are indicators of the width of the previous layers (*e.g.*, the output vector length of a fully-connected layer or the number of filters of a convolutional layer). The matchings between the nodes are marked by light blue. So a matching between the nodes can be seen as a matching between the layers. To morph f_a to f_b with the given matching, we need to first widen the three nodes in f_a to the same width as their matched nodes in f_b, and then insert a new node of width 20 after the first node in f_a. Based on this morphing scheme, the overall edit-distance of the layers is defined as D_l in Equation (4).

Since there are many ways to morph f_a to f_b, to find the best matching between the nodes that minimizes D_l, we propose a dynamic programming approach by defining a matrix $\boldsymbol{A}_{|L_a|\times|L_b|}$, which is recursively calculated as follows:

$$\boldsymbol{A}_{i,j} = max[\boldsymbol{A}_{i-1,j}+1, \boldsymbol{A}_{i,j-1}+1, \boldsymbol{A}_{i-1,j-1}+d_l(l_a^{(i)}, l_b^{(j)})], \tag{6}$$

where $\boldsymbol{A}_{i,j}$ is the minimum value of $D_l(L_a^{(i)}, L_b^{(j)})$, where $L_a^{(i)} = \{l_a^{(1)}, l_a^{(2)}, \dots, l_a^{(i)}\}$ and $L_b^{(j)} = \{l_b^{(1)}, l_b^{(2)}, \dots, l_b^{(j)}\}$.

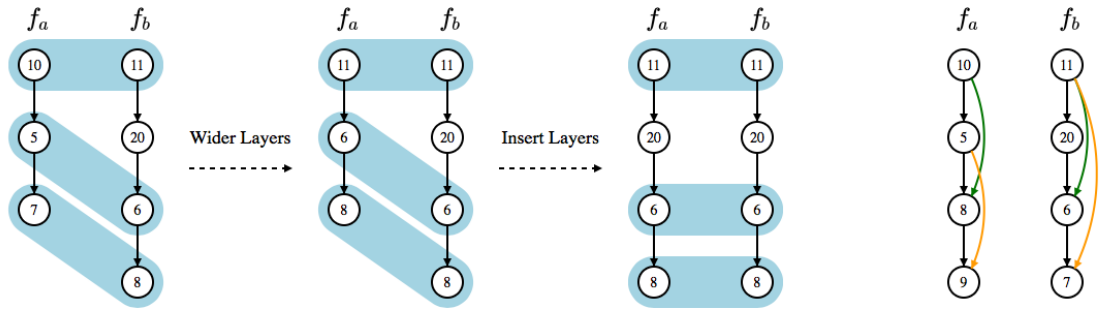


Figure 1. Neural Network Kernel. Given two neural networks f_a, f_b, and matchings between the similar layers, the figure shows how the layers of f_a can be changed to the same as f_b. Similarly, the skip-connections in f_a also need to be changed to the same as f_b according to a given matching.

Calculating D_s: The intuition of D_s is the sum of the the edit-distances of the matched skip-connections in two neural networks into pairs. As shown in Figure 1, the skip-connections with the same color are matched pairs. Similar to $D_l(\cdot,\cdot)$, $D_s(\cdot,\cdot)$ is defined as follows:

$$D_s(S_a, S_b) = \min \sum_{i=1}^{|S_a|} d_s(s_a^{(i)}, \varphi_s(s_a^{(i)})) + \Big| |S_b| - |S_a| \Big|, \tag{7}$$

where we assume $|S_a| < |S_b|$. $(|S_b| - |S_a|)$ measures the total edit-distance for non-matched skip-connections since each of the non-matched skip-connections in S_b calls for an edit of inserting a new skip connection into f_a. The mapping function $\varphi_s : S_a \to S_b$ is an injective function. $d_s(\cdot,\cdot)$ is the edit-distance for two matched skip-connections defined as:

$$d_s(s_a, s_b) = \frac{|u(s_a) - u(s_b)| + |\delta(s_a) - \delta(s_b)|}{max[u(s_a), u(s_b)] + max[\delta(s_a), \delta(s_b)]}, \tag{8}$$

where $u(s)$ is the topological rank of the layer the skip-connection s started from, $\delta(s)$ is the number of layers between the start and end point of the skip-connection s.

This minimization problem in Equation (7) can be mapped to a bipartite graph matching problem, where f_a and f_b are the two disjoint sets of the graph, each skip-connection is a node in its corresponding set. The edit-distance between two skip-connections is the weight of the edge between them. The bipartite graph problem is solved by Hungarian algorithm (Kuhn-Munkres algorithm) (Kuhn, 1955).

Proof of Kernel Validity: Gaussian process requires the kernel to be valid, *i.e.*, the kernel matrices are positive semidefinite, to keep the distributions valid. The edit-distance in Equation (3) is a metric distance proved by Theorem 1. Though, a generalized RBF kernel in the form of $e^{-\gamma d(x,y)}$ based on a distance in metric space may not always be a valid kernel, our kernel defined in Equation (2) is proved to be valid by Theorem 2.

Theorem 1. $d(f_a, f_b)$ is a metric space distance.

Proof of Theorem 1: See appendix. $\square$

Theorem 2. $\kappa(f_a, f_b)$ is a valid kernel.

Proof of Theorem 2: The kernel matrix of generalized RBF kernel in the form of $e^{-\gamma D^2(x,y)}$ is positive definite if and only if there is an isometric embedding in Euclidean space for the metric space with metric D (Haasdonk & Bahlmann, 2004). Any finite metric space distance can be isometrically embedded into Euclidean space by changing the scale of the distance measurement (Maehara, 2013). By using Bourgain theorem (Bourgain, 1985), metric space d is embedded to Euclidean space with little distortion. $\rho(d(f_a, f_b))$ is the embedded distance for $d(f_a, f_b)$. Therefore, $e^{-\rho^2(d(f_a,f_b))}$ is always positive definite. So $\kappa(f_a, f_b)$ is a valid kernel. $\square$

3.2 Acquisition Function Optimization for Tree Structured Space

The second challenge is the optimization of the acquisition function. The traditional acquisition functions are defined on Euclidean space. The methods optimizing them are not applicable to the tree-structured search via network morphism. To deal with this problem, we propose a novel method to optimize the acquisition function for tree-structured space.

Upper-confidence bound (UCB) (Auer et al., 2002) is selected as our acquisition function to be minimized. The value of UCB function is an estimation of the lowest possible value of the cost function given the neural network f defined as follows

$$\alpha(f) = \mu(y_f) - \beta\sigma(y_f), \tag{9}$$

where $y_f = \min_{\boldsymbol{\theta_f}} Cost(f(\boldsymbol{X}; \boldsymbol{\theta_f}))$, β is the balancing factor, $\mu(y_f)$ and $\sigma(y_f)$ are the posterior mean and standard deviation of variable y_f. UCB has two important properties fit our problem. First, it has an explicit balance factor β for exploration and exploitation. Second, the function value is directly comparable with the cost function value $c^{(i)}$ in search history $\mathcal{H} = \{(f^{(i)}, \boldsymbol{\theta}^{(i)}, c^{(i)})\}$. With the acquisition function, $\hat{f} = argmin_f \alpha(f)$ is the generated architecture for next observation.

The tree-structured space is defined as follows. During the optimization of the $\alpha(f)$, $\hat{f}$ should be obtained from $f^{(i)}$ and O, where $f^{(i)}$ is an observed architecture in the search history $\mathcal{H}$, O is a sequence of operations to morph the architecture into a new one. Morph f to $\hat{f}$ with O is denoted as $\hat{f} \leftarrow \mathcal{M}(f, O)$, where $\mathcal{M}(\cdot, \cdot)$ is the function to morph f with the operations in O. Therefore, the search can be viewed as a tree-structured search, where each node is a neural architecture, whose children are morphed from it by network morphism operations.

The most common defect of network morphism is it only grows the size of the architecture instead of shrinking them. Some other work using network morphism for NAS (Elsken et al., 2018) would end up with a very large architecture without enough exploration on the smaller architectures. The main reason is they only morph the current best architectures instead of the smaller architectures found earlier in the search. However, in our tree-structure search, we can not only expand the leaves but also the inner nodes, which means the smaller architectures found in the early stage can be selected multiple times to morph to more architectures.

The state-of-the-art acquisition function optimization techniques, *e.g.*, gradient-based or Newton-like method, are designed for numerical data, which do not apply in the tree-structure space. TreeBO (Jenatton et al., 2017) has proposed a way to maximize the acquisition function in a tree-structured parameter space. However, only the leaf nodes of its tree have acquisition function values, which is different from our case. Moreover, the proposed solution is a surrogate multivariate Bayesian optimization model. In NASBOT (Kandasamy et al., 2018), they use an evolutionary algorithm to optimize the acquisition function. They are both very expansive in computing time. To optimize our acquisition function, we need a method to efficiently optimize the acquisition function in the tree-structured space.

Inspired by the various heuristic search algorithms for exploring the tree-structured search space and optimization methods balancing between exploration and exploitation, a new method based on A* search and simulated annealing is proposed. A* is for tree-structure search. Simulated annealing is for balancing exploration and exploitation.

As shown in Algorithm 1, the algorithm takes minimum temperature T_{low}, temperature decreasing rate r for simulated annealing, and search history $\mathcal{H}$ described in Section 2 as input. It outputs a neural architecture $f \in \mathcal{H}$ and a sequence of operations O to morph f into the new architecture. From line 2 to line 6, the searched architectures are pushed into

Algorithm 1 Optimize Acquisition Function

```
1:  Input: $\mathcal{H}$, $r$, $T_{low}$
2:  $T \leftarrow 1$, $Q \leftarrow PriorityQueue()$
3:  $c_{min} \leftarrow$ lowest $c$ in $\mathcal{H}$
4:  for $(f, \boldsymbol{\theta_f}, c) \in \mathcal{H}$ do
5:      $Q.Push(f)$
6:  end for
7:  while $Q \neq \varnothing$ and $T > T_{low}$ do
8:      $T \leftarrow T \times r$, $f \leftarrow Q.Pop()$
9:      for $o \in \Omega(f)$ do
10:         $f' \leftarrow \mathcal{M}(f, \{o\})$
11:         if $e^{\frac{c_{min} - \alpha(f')}{T}} > Rand()$ then
12:             $Q$.Push($f'$)
13:         end if
14:         if $c_{min} > \alpha(f')$ then
15:             $c_{min} \leftarrow \alpha(f')$, $f_{min} \leftarrow f'$
16:         end if
17:     end for
18: end while
19: Return The nearest ancestor of $f_{min}$ in $\mathcal{H}$, the operation sequence to reach $f_{min}$
```

the priority queue, which sorts the elements according to the cost function value or the acquisition function value. Since UCB is chosen as the acquisiton function, $\alpha(f)$ is directly comparable with the history observation values $c^{(i)}$. From line 7 to line 18 is the loop optimizing the acquisition function. Following the setting in A* search, in each iteration, the architecture with the lowest acquisition function value is pop out to be expanded on line 8 to 10, where $\Omega(f)$ is all the possible operations to morph the architecture f, $\mathcal{M}(f, o)$ is the function to morph the architecture f with the operation sequence o. However, not all the children are pushed into the priority queue for exploration purpose. The decision is made by simulated annealing on line 11, where $e^{\frac{c_{min} - \alpha(f')}{T}}$ is a typical acceptance function in simulated annealing. c_{min} and f_{min} are updated on from line 14 to 16, which record the minimum acquisition function value and the corresponding architecture.

3.3 Graph-Level Network Morphism Operations

The third challenge is to maintain the intermediate output tensor shape consistency when morphing the architectures. Previous work showed how to preserve the functionality of the layers the operators applied on, namely layer-level morphism. However, from a graph-level view, any change of a single layer could have a butterfly effect on the entire network. Otherwise, it would break the input and output tensor shape consistency. To tackle the challenge, a graph-level morphism is proposed to find and morph the layers influenced by a layer-level operation in the entire network.

There are four network morphism operations we could perform on a neural network $f \in \mathcal{F}$ (Elsken et al., 2017), which can all be reflected in the change of the computational graph G. The first operation is inserting a layer to f to make it deeper denoted as $deep(G, u)$, where u is the node marking the place to insert the layer. The second one is widening a node in f denoted as $wide(G, u)$, where u is the node representing the intermediate output tensor to be widened. Widen here could be either making the output vector of the previous fully-connected layer of u longer, or adding more filters to the previous convolutional layer of u, depending on the type of the previous layer. The third one is adding an additive connection from node u to node v denoted as $add(G, u, v)$. The fourth one is adding an concatenative connection from node u to node v denoted as $concat(G, u, v)$. For $deep(G, u)$, no other operation is needed except for initializing the weights of the newly added layer as described in (Chen et al., 2015). However, for all other three operations, more changes are required to G.

First, we define an effective area of $wide(G, u_0)$ as γ to better describe where to change in the network. The effective area is a set of nodes in the computational graph, which can be recursively defined by the following rules: 1. $u_0 \in \gamma$. 2. $v \in \gamma$, if $\exists e_{u \to v} \notin L_s$, $u \in \gamma$. 3. $v \in \gamma$, if $\exists e_{v \to u} \notin L_s$, $u \in \gamma$. L_s is the set of fully-connected layers and convolutional layers. Operation $wide(G, u_0)$ needs to change two set of layers, the previous layer set $L_p = \{e_{u \to v} \in L_s | v \in \gamma\}$, which needs to output a wider tensor, and next layer set $L_n = \{e_{u \to v} \in L_s | u \in \gamma\}$, which needs to input a wider tensor. Second, for operator $add(G, u_0, v_0)$, additional pooling layers may be needed on the skip-connection. u_0 and v_0 have the same number of channels, but their shape may differ because of the pooling layers between them. So we need a set of pooling layers whose effect is the same as the combination of all the pooling layers between u_0 and v_0, which is defined as $L_o = \{e \in L_{pool} | e \in p_{u_0 \to v_0}\}$. where $p_{u_0 \to v_0}$ could be any path between u_0 and v_0, L_{pool} is the pooling layer set. Another layer L_c is used after to pooling layers to process u_0 to the same width as v_0. Third, in $concat(G, u_0, v_0)$, the concatenated tensor is wider than the original tensor v_0. The concatenated tensor is input to a new layer L_c to reduce the width back to the same width as v_0. Additional pooling layers are also needed for the concatenative connection.

3.4 Architecture Performance Estimation

During the observation, we need to estimate the performance of a neural architecture to update the Gaussian process model in Bayesian optimization. Although, various performance estimation strategies (Brock et al., 2017; Pham et al., 2018; Elsken et al., 2018) are available to speed up the process, the most accurate way to observe the performance of a neural architecture is to actually train it. The quality of the observations is essential to the neural architecture search algorithm. So the neural architectures are actually trained during the search in our proposed method.

There two important requirements for the training process. First, it needs to be adaptive to different architectures. Different neural networks require different numbers of epochs in training to converge. Second, it should not be affected by the noise in the performance curve. The final metric value, *e.g.*, mean squared error or accuracy, on the validation set is not the best performance estimation since there is random noise in it.

To be adaptive to architectures of different sizes, we use the same strategy as the early stop strategy in the multi-layer perceptron algorithm in Scikit-Learn (Pedregosa et al., 2011). It sets a maximum threshold τ. If the loss of the validation set does not decrease in τ epochs, the training stops. Comparing with many state-of-the-art methods using a fixed number of training epochs, it is more adaptive to different architectures.

To be not affected by the noise in the performance, the mean of metric values of the last τ epochs on the validation set is used as the estimated performance for the given neural architecture. It is more accurate than using the final metric value on the validation set.

3.5 Time Complexity Analysis

As described at the start of Section 3 in the paper, Bayesian optimization can be roughly divided into three steps: update, generation, and observation. The bottle-neck of the efficiency of the algorithm is observation, which involves the training of the generated neural architecture. Let n be the number of architectures in the search history. The time complexity of the update is $O(n^2 \log_2 n)$. In each generation, the kernel is computed between the new architectures during optimizing acquisition function and the ones in the search history, the number of values in which is $O(nm)$, where m is the number of architectures computed during the optimization of the acquisition function. The time complexity for computing $d(\cdot, \cdot)$ once is $O(l^2 + s^3)$, where l and s are the number of layers and skip-connections. So the overall time complexity is $O(nm(l^2 + s^3) + n^2 \log_2 n)$. The magnitude of these factors is within the scope of tens. So the time consumption of update and generation is trivial comparing to the observation time.

4 AUTO-KERAS

Based on the proposed neural architecture search method, an open-source AutoML system, namely Auto-Keras, is developed. It is named after Keras (Chollet et al., 2015), which is famous for its simplicity in creating neural networks. Similar to SMAC (Hutter et al., 2011), Auto-WEKA (Thornton et al., 2013), and Auto-Sklearn (Feurer et al., 2015), the

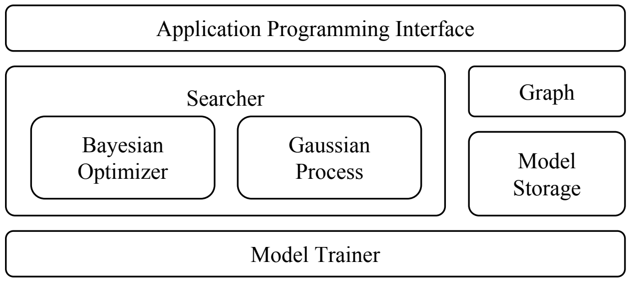


Figure 2. Auto-Keras System Overview

goal is to enable domain experts who are not familiar with machine learning technologies to use the machine learning techniques easily. However, Auto-Keras is focusing on the deep learning tasks, which is different from the systems focusing on the shallow models mentioned above. Although, there are several AutoML services available on large cloud computing platforms, three things are prohibiting the users from using them. First, the cloud services are not free to use, which may not be affordable for everyone who wants to use AutoML techniques. Second, the cloud services based AutoML usually requires complicated configurations of Docker containers and Kubernetes, which is not easy for people without computer science background. Third, the AutoML service providers are honest but curious, which cannot guarantee the security and privacy of the data. An open-source software, which is easily downloadable and runs locally, would solve these problems and make the AutoML accessible to everyone. To address such need, we developed Auto-Keras. The system is carefully designed with an easy-to-use application programming interface (API), CPU and GPU parallelism, and GPU memory adaption.

4.1 Overview

The Auto-Keras system consists of three major parts from the top down as shown in Figure 2. The top part is the API, which is directly called by the users. It is responsible for calling corresponding middle-level modules to complete certain functionalities. The middle part of the system consists of three modules. The Searcher is the module containing Bayesian Optimizer and Gaussian Process. The Graph is the module processing the computational graphs of neural networks, which has functions implemented for the network morphism operations. The Model Storage is a pool of trained models. Since the size of the neural networks are large and cannot be stored all in memory, the model storage saves all the trained models on disk. At the bottom of the figure is Model Trainer, which trains given neural networks with the training data in a separate process for parallelism.

A typical workflow for the Auto-Keras system would be as follows. the user initiated a search for the best neural architecture for the dataset. The API received the call, preprocess the dataset, and pass it to the Searcher to start the search.

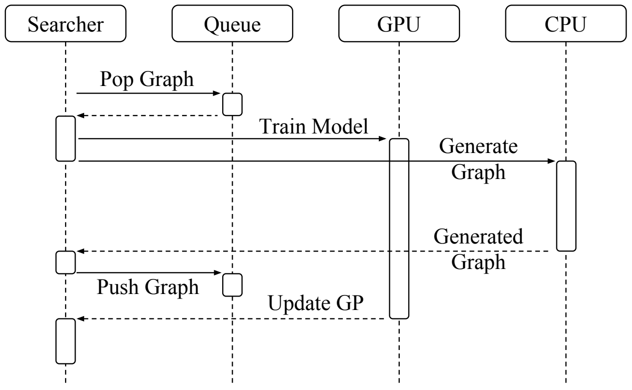


Figure 3. Parallel Sequence Diagram

The Bayesian Optimizer in the Searcher would generate a new architecture. It calls the Graph module to morph an existing model in the Model Storage into the new one. The new model is passed to the Model Trainer to train with the dataset. The trained model is saved in the Model Storage. The performance of the model is feedback to the Searcher to update the Gaussian Process.

4.2 Interface

The design of the API follows the classic design of the Scikit-Learn API (Pedregosa et al., 2011; Buitinck et al., 2013). The training of a neural network requires as few as three lines of code calling the constructor, the fit and predict function respectively. Users can also specify the model trainer's hyperparameters using the default parameters to the functions. Several accommodations have been implemented to enhance the user experience with the Auto-Keras package. First, the user can restore and continue a previous search which might be accidentally killed. From the users' perspective, the main difference of using Auto-Keras comparing with the AutoML systems aiming at shallow models is the much longer time consumption, since a number of deep neural networks are trained during the neural architecture search. It is possible for some accident to happen to kill the process before the search finishes. Therefore, the search outputs all the searched neural network architectures with their trained parameters into a specific directory on the disk. As long as the path to the directory is provided, the previous search can be restored. Second, the user can export the search results, which are neural architectures, as saved Keras models for other usages. Third, for advanced users, they can specify all kinds of hyperparameters of the search process and neural network optimization process by the default parameters in the interface.

4.3 Parallelism

The program can run in parallel on the GPU and the CPU at the same time. If we do the observation (training of the current neural network), update, and generation of Bayesian optimization in an sequential order. The GPUs will be idle during the update and generation. The CPUs will be idle during the observation. To improve the efficiency, the observation is run in parallel with the generation in separated processes. A training queue is maintained as a buffer for the Model Trainer. Figure 3 shows the Sequence diagram of the parallelism between the CPU and the GPU. First, the Searcher requests the queue to pop out a new graph and pass it to GPU to start training. Second, while the GPU is busy, the searcher request the CPU to generate a new graph. At this time period, the GPU and the CPU work in parallel. Third, the CPU returns the generated graph to the searcher, who pushes the graph into the queue. Finally, the Model Trainer finished training the graph on the GPU and returns it to the Searcher to update the Gaussian process. In this way, the idle time of GPU and CPU are dramatically reduced to improve the efficiency of the search process.

4.4 GPU Memory Adaption

Since different deployment environments of Auto-Keras have different limitations on the GPU memory usage, the size of the neural networks needs to be limited according to the GPU memory limitation. Otherwise, the system would crash because of running out of GPU memory. To tackle this challenge, we implemented a memory estimation function on our own data structure for the neural architectures. An integer value is used to mark the upper bound of the neural architecture size. Any new computational graph whose estimated size exceeds the upper bound is discarded. However, the system may still crash because the management of the GPU memory is very complicated, which cannot be precisely estimated. So whenever it runs out of GPU memory, the upper bound is lowered down to further limit the size of the neural networks generated.

5 Experiments

In the experiments, we aim at answering the following questions. 1) How effective is the search algorithm with limited running time? 2) How much efficiency are gained from Bayesian optimization and network morphism? 3) What are the influences of the important hyperparameters of the search algorithm? 4) Does the proposed kernel function correctly measure the similarity among neural networks in their actual performance?

Datasets Three benchmark datasets, MNIST (LeCun et al., 1998), CIFAR10 (Krizhevsky & Hinton, 2009), and FASHION (Xiao et al., 2017) are used in the experiments to evaluate our method. They prefer very different neural architectures to achieve good performance.

Baselines Four categories of baseline methods are used for comparison, which are elaborated as follows:

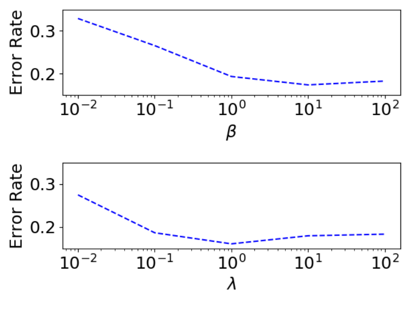
- Straightforward Methods: random search (RAND) and grid search (GRID). They search the number of layers and the width of the layers.
- Conventional Methods: SPMT (Snoek et al., 2012) and SMAC (Hutter et al., 2011). Both SPMT and SMAC are designed for general hyperparameters tuning tasks of machine learning models instead of focusing on the deep neural networks. They tune the 16 hyperparameters of a three-layer convolutional neural network, including the width, dropout rate, and regularization rate of each layer.
- State-of-the-art Methods: SEAS (Elsken et al., 2017), NASBOT (Kandasamy et al., 2018). We carefully implemented the SEAS as described in their paper. For NASBOT, since the experimental settings are very similar, we directly trained their searched neural architecture in the paper. They did not search architectures for MNIST and FASHION dataset, so the results are omitted in our experiments.
- Variants of the proposed method: BFS and BO. our proposed method is denoted as AK. BFS replaces the Bayesian optimization in AK with the breadth-first search. BO is another variant, which does not employ network morphism to speed up the training. For AK, β is set to 2.5, while λ is set to 1 according to the parameter sensitivity analysis experiment.

Experimental Setting The general experimental setting for evaluation is described as follows: First, the original training data of each dataset is further divided into training and validation sets by 80-20. Second, the testing data of each dataset is used as the testing set. Third, the initial architecture for SEAS, BO, BFS, and AK is a three-layer convolutional neural network with 64 filters in each layer. Fourth, each method is run for 12 hours on a single GPU (NVIDIA GeForce GTX 1080 Ti) on the training and validation set with batch size of 64. Fifth, the output architecture is trained with both the training and validation set. Sixth, the testing set is used to evaluate the trained architecture. Error rate is selected as the evaluation metric since all the datasets are for classification. For a fair comparison, the same data processing and training procedures are used for all the methods. The neural networks are trained for 200 epochs in all the experiments.

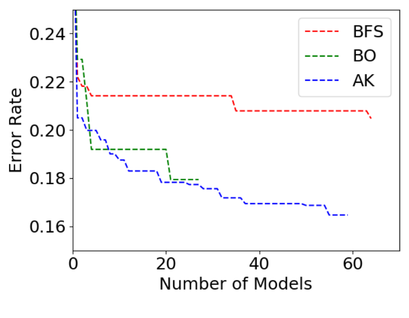
5.1 Evaluation of Effectiveness

We first evaluate the effectiveness of the proposed method. The results are shown in Table 1. Four main conclusions could be drawn based on the results.

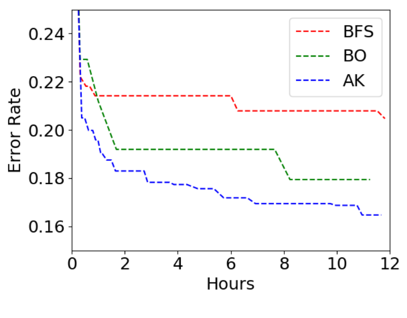
(1) The proposed method AK achieves the lowest error rate on all the three datasets. It demonstrates that AK is able



(a) Parameter Analysis



(b) Performance and Number of Models



(c) Performance and Time

Figure 4. Parameter Analysis and Evaluation of Efficiency

Table 1. Classification error rate

Methods	MNIST	CIFAR10	FASHION
RANDOM	1.79%	16.86%	11.36%
GRID	1.68%	17.17%	10.28%
SPMT	1.36%	14.68%	9.62%
SMAC	1.43%	15.04%	10.87%
SEAS	1.07%	12.43%	8.05%
NASBOT	NA	12.30%	NA
BFS	1.56%	13.84%	9.13%
BO	1.83%	12.90%	7.99%
AK	**0.55%**	**11.44%**	**7.42%**

to search out simple but effective architectures on small datasets (*e.g.*, MNIST) as well as exploring more complicated structures on larger datasets (*e.g.*, CIFAR10).

(2) The straightforward approaches and traditional approaches perform well on the MNIST dataset, but poorly on the CIFAR10 dataset. This may come from the fact that: naive approaches like random search and grid search only try a limited number of architectures blindly while the two conventional approaches are unable to change the depth and skip-connections of the architectures.

(3) Though the two state-of-the-art approaches achieve acceptable performance, SEAS could not beat our proposed model due to its subpar search strategy. The hill-climbing strategy it adopts only takes one step at each time in morphing the current best architecture, and the search tree structure is constrained to be unidirectionally extending. Comparatively speaking, NASBOT possesses stronger search expandability and also uses Bayesian optimization as our proposed method. However, the low efficiency constrains its power in achieving comparable performance within a short time period. By contrast, the network morphism scheme along with the novel searching strategy ensures our model to achieve desirable performance with limited hardware resources and time budges.

(4) For the two variants of AK, BFS preferentially considers searching a vast number of neighbors surrounding the initial architecture, which constrains its power in reaching the better architectures away from the initialization. By comparison, BO can jump far from the initial architecture. But without network morphism, it needs to train each neural architecture with much longer time, which limits the number of architectures it can search within a given time.

5.2 Evaluation of Efficiency

In this experiment, we try to evaluate the efficiency gain of the proposed method in two aspects. First, we evaluate whether Bayesian optimization can really find better solutions with a limited number of observations. Second, we evaluated whether network morphism can enhance the training efficiency.

We compare the proposed method AK with its two variants, BFS and BO, to show the efficiency gains from Bayesian optimization and network morphism, respectively. BFS does not adopt Bayesian optimization but only network morphism, and use breadth-first search to select the network morphism operations. BO does not employ network morphism but only Bayesian optimization. Each of the three methods is run on CIFAR10 for twelve hours. Figure 4b shows the relation between the lowest error rate achieved and the number of neural networks searched. Figure 4c shows the relation between the lowest error rate achieved and the searching time.

Two conclusions can be drawn by comparing BFS and AK. First, Bayesian optimization can efficiently find better architectures with a limited number of observations. When searched the same number of neural architectures, AK could achieve a much lower error rate than BFS. It demonstrates that Bayesian optimization could effectively guide the search in the right direction, which is much more efficient in finding good architectures than the naive BFS approach. Second, the overhead created by Bayesian optimization during the search is low. In Figure 4b, it shows BFS and AK searched similar numbers of neural networks within twelve hours. BFS is a naive search strategy, which does not consume much time during the search besides training the neural networks. AK searched slightly less number of neural architectures than BFS because it takes more time to run.

Two conclusions can be drawn by comparing BO and AK. First, network morphism does not negatively impact the search performance. In Figure 4b, when BO and AK search a similar number of neural architectures, they achieve similar lowest error rates. Second, network morphism increases the training efficiency, thus improve the performance. As shown in Figure 4b, AK could search much more architectures than BO within the same amount of time due to the adoption of network morphism. Since network morphism does not degrade the search performance, searching more architectures results in finding better architectures. This could also be confirmed in Figure 4c. At the end of the searching time, AK achieves lower error rate than BO.

5.3 Parameter Sensitivity Analysis

We now analyze the impacts of the two most important hyperparameters in our proposed method, *i.e.*, λ in Equation (3) and β in Equation (9). λ balances the distance of layers and skip connections in the kernel function, and β is the weight of the variance in the acquisition function, which balances the exploration and exploitation of the search strategy. Since r and T_{low} in Algorithm 1 are just normal hyperparameters of simulated annealing, we do not delve into them here. The dataset used in this section is CIFAR10. The other experimental setup here follows the former settings.

From Figure 4a, we can observe that the influences of β and λ to the performance of our method are similar. As shown in the upper part of Figure 4a, with the increase of β from 10^{-2} to 10^{2}, the error rate decreases first and then increases. If β is too small, the search process is not explorative enough to search the architectures far from the initial architecture. If it is too large, the search process would keep exploring the far points instead of trying the most promising architectures. Similarly, as shown in the lower part of Figure 4a, the increase of λ would downgrade the error rate at first and then upgrade it. This is because if λ is too small, the differences in the skip-connections of two neural architectures are ignored; conversely, if it is too large, the differences in the convolutional or fully-connected layers are ignored. The differences in layers and skip-connections should be balanced in the kernel function for the entire framework to achieve a good performance.

5.4 Evaluation of Kenrel Quality

To show the quality of the edit-distance neural network kernel, we investigate the difference between the two matrices $\boldsymbol{K}$ and $\boldsymbol{P}$. $\boldsymbol{K}_{n\times n}$ is the kernel matrix, where $\boldsymbol{K}_{i,j} = \kappa(f^{(i)}, f^{(j)})$. $\boldsymbol{P}_{n\times n}$ describes the similarity of

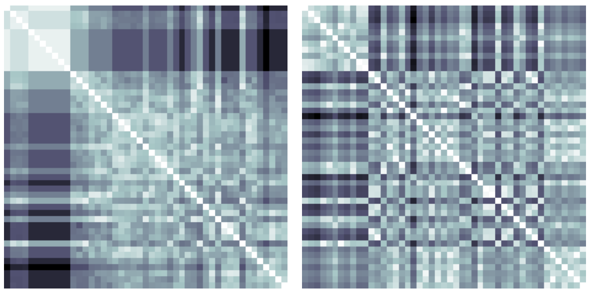

(a) Kernel matrix (b) Performance matrix

Figure 5. Kernel and performance matrix visualization

the actual performance between neural networks, where $\boldsymbol{P}_{i,j} = -|c^{(i)} - c^{(j)}|$, $c^{(i)}$ is the cost function value in the search history $\mathcal{H}$ described in Section 3. We use CIFAR10 as an example here, and adopt error rate as the cost metric. Since the values in $\boldsymbol{K}$ and $\boldsymbol{P}$ are in different scales, both matrices are normalized to the range $[-1, 1]$. We quantitatively measure the difference between $\boldsymbol{K}$ and $\boldsymbol{P}$ with mean square error, which is 1.12×10^{-1}.

$\boldsymbol{K}$ and $\boldsymbol{P}$ are visualized in Figure 5a and 5b. Lighter color means larger values. There are two patterns can be observed in the figures.

First, the white diagonal of Figure 5a and 5b. According to the definiteness property of the kernel, $\kappa(f_x, f_x) = 1, \forall f_x \in \mathcal{F}$, thus the diagonal of $\boldsymbol{K}$ is always 1. It is the same for $\boldsymbol{P}$ since no difference exists in the performance of the same neural network.

Second, there is a small light square area on the upper left of Figure 5a. These are the initial neural architectures to train the Bayesian optimizer, which are neighbors to each other in terms of network morphism operations. A similar pattern is reflected in Figure 5b, which indicates that when the kernel measures two architectures as similar, they tend to have similar performance.

6 Conclusion and Future Work

In this paper, a novel method for efficient neural architecture search with network morphism is proposed. It enables Bayesian optimization to guide the search by designing a neural network kernel, and an algorithm for optimizing acquisition function in tree-structured space. The proposed method is wrapped into an open-source AutoML system, namely Auto-Keras, which can be easily downloaded and used with an extremely simple interface. The method has shown good performance in the experiments and outperformed several traditional hyperparameter-tuning methods and state-of-the-art neural architecture search methods. In the future, the search space may be expanded to the recurrent neural network (RNN). It is also important to tune the neural architecture and the hyperparameters of the training process together to further save the manual labor.

References

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

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

Bourgain, J. On lipschitz embedding of finite metric spaces in hilbert space. *Israel Journal of Mathematics*, 52(1-2): 46–52, 1985.

Brock, A., Lim, T., Ritchie, J. M., and Weston, N. Smash: one-shot model architecture search through hypernetworks. *arXiv preprint arXiv:1708.05344*, 2017.

Buitinck, L., Louppe, G., Blondel, M., Pedregosa, F., Mueller, A., Grisel, O., Niculae, V., Prettenhofer, P., Gramfort, A., Grobler, J., Layton, R., VanderPlas, J., Joly, A., Holt, B., and Varoquaux, G. API design for machine learning software: experiences from the scikit-learn project. In *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*, pp. 108–122, 2013.

Cai, H., Chen, T., Zhang, W., Yu, Y., and Wang, J. Efficient architecture search by network transformation. In *AAAI*, 2018.

Chen, T., Goodfellow, I., and Shlens, J. Net2net: Accelerating learning via knowledge transfer. *arXiv preprint arXiv:1511.05641*, 2015.

Chollet, F. et al. Keras. https://keras.io, 2015.

Desell, T. Large scale evolution of convolutional neural networks using volunteer computing. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, pp. 127–128. ACM, 2017.

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

Elsken, T., Metzen, J. H., and Hutter, F. Neural architecture search: A survey. *arXiv preprint arXiv:1808.05377*, 2018.

Feurer, M., Klein, A., Eggensperger, K., Springenberg, J., Blum, M., and Hutter, F. Efficient and robust automated machine learning. In *Advances in Neural Information Processing Systems*, pp. 2962–2970, 2015.

Haasdonk, B. and Bahlmann, C. Learning with distance substitution kernels. In *Joint Pattern Recognition Symposium*, pp. 220–227. Springer, 2004.

Hutter, F., Hoos, H. H., and Leyton-Brown, K. Sequential model-based optimization for general algorithm configuration. *LION*, 5:507–523, 2011.

Jenatton, R., Archambeau, C., González, J., and Seeger, M. Bayesian optimization with tree-structured dependencies. In *International Conference on Machine Learning*, pp. 1655–1664, 2017.

Kandasamy, K., Neiswanger, W., Schneider, J., Poczos, B., and Xing, E. Neural architecture search with bayesian optimisation and optimal transport. *NIPS*, 2018.

Kotthoff, L., Thornton, C., Hoos, H. H., Hutter, F., and Leyton-Brown, K. Auto-weka 2.0: Automatic model selection and hyperparameter optimization in weka. *Journal of Machine Learning Research*, 17:1–5, 2016.

Krizhevsky, A. and Hinton, G. Learning multiple layers of features from tiny images. 2009.

Kuhn, H. W. The hungarian method for the assignment problem. *Naval Research Logistics (NRL)*, 2(1-2):83–97, 1955.

LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

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

Maehara, H. Euclidean embeddings of finite metric spaces. *Discrete Mathematics*, 313(23):2848–2856, 2013.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Duchesnay, E. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

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

Real, E., Moore, S., Selle, A., Saxena, S., Suematsu, Y. L., Le, Q., and Kurakin, A. Large-scale evolution of image classifiers. *arXiv preprint arXiv:1703.01041*, 2017.

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

Snoek, J., Larochelle, H., and Adams, R. P. Practical bayesian optimization of machine learning algorithms. In *Advances in Neural Information Processing Systems*, pp. 2951–2959, 2012.

Suganuma, M., Shirakawa, S., and Nagao, T. A genetic programming approach to designing convolutional neural network architectures. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pp. 497–504. ACM, 2017.

Thornton, C., Hutter, F., Hoos, H. H., and Leyton-Brown, K. 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*, pp. 847–855. ACM, 2013.

Wei, T., Wang, C., Rui, Y., and Chen, C. W. Network morphism. In *International Conference on Machine Learning*, pp. 564–572, 2016.

Xiao, H., Rasul, K., and Vollgraf, R. Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms, 2017.

Xie, L. and Yuille, A. Genetic cnn. *arXiv preprint arXiv:1703.01513*, 2017.

Yanardag, P. and Vishwanathan, S. Deep graph kernels. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 1365–1374. ACM, 2015.

Zeng, Z., Tung, A. K., Wang, J., Feng, J., and Zhou, L. Comparing stars: On approximating graph edit distance. *Proceedings of the VLDB Endowment*, 2(1):25–36, 2009.

Zhong, Z., Yan, J., and Liu, C.-L. Practical network blocks design with q-learning. *arXiv preprint arXiv:1708.05552*, 2017.

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

A APPENDIX

Theorem 1. $d(f_a, f_b)$ is a metric space distance.

Proof of Theorem 1: Theorem 1 is proved by proving the non-negativity, definiteness, symmetry, and triangle inequality of d.

Non-negativity:

$\forall f_x\ f_y \in \mathcal{F}, d(f_x, f_y) \geq 0$.

From the definition of $w(l)$ in Equation (5), $\forall l$, $w(l) > 0$. $\therefore \forall l_x\ l_y, d_l(l_x, l_y) \geq 0$. $\therefore \forall L_x\ L_y, D_l(L_x, L_y) \geq 0$. Similarly, $\forall s_x\ s_y, d_s(s_x, s_y) \geq 0$, and $\forall S_x\ S_y, D_s(S_x, S_y) \geq 0$. In conclusion, $\forall f_x\ f_y \in \mathcal{F}, d(f_x, f_y) \geq 0$.

Definiteness:

$f_a = f_b \iff d(f_a, f_b) = 0$.

$f_a = f_b \implies d(f_a, f_b) = 0$ is trivial. To prove $d(f_a, f_b) = 0 \implies f_a = f_b$, let $d(f_a, f_b) = 0$. $\because \forall L_x\ L_y$, $D_l(L_x, L_y) \geq 0$ and $\forall S_x\ S_y$, $D_s(S_x, S_y) \geq 0$. Let L_a and L_b be the layer sets of f_a and f_b. Let S_a and S_b be the skip-connection sets of f_a and f_b.

$\therefore D_l(L_a, L_b) = 0$ and $D_s(S_a, S_b) = 0$. $\because \forall l_x\ l_y$, $d_l(l_x, l_y) \geq 0$ and $\forall s_x\ s_y$, $d_s(s_x, s_y) \geq 0$. $\therefore |L_a| = |L_b|$, $|S_a| = |S_b|$, $\forall l_a \in L_a$, $l_b = \varphi_l(l_a) \in L_b$, $d_l(l_a, l_b) = 0$, $\forall s_a \in S_a$, $s_b = \varphi_s(s_a) \in S_b$, $d_s(s_a, s_b) = 0$. According to Equation (5), each of the layers in f_a has the same width as the matched layer in f_b, According to the restrictions of $\varphi_l(\cdot)$, the matched layers are in the same order, and all the layers are matched, i.e. the layers of the two networks are exactly the same. Similarly, the skip-connections in the two neural networks are exactly the same. $\therefore f_a = f_b$. So $d(f_a, f_b) = 0 \implies f_a = f_b$, let $d(f_a, f_b) = 0$. Finally, $f_a = f_b \iff d(f_a, f_b)$.

Symmetry:

$\forall f_x\ f_y \in \mathcal{F}, d(f_x, f_y) = d(f_y, f_x)$.

Let f_a and f_b be two neural networks in $\mathcal{F}$, Let L_a and L_b be the layer sets of f_a and f_b. If $|L_a| \neq |L_b|$, $D_l(L_a, L_b) = D_l(L_b, L_a)$ since it will always swap L_a and L_b if L_a has more layers. If $|L_a| = |L_b|$, $D_l(L_a, L_b) = D_l(L_b, L_a)$ since $\varphi_l(\cdot)$ is undirected, and $d_l(\cdot, \cdot)$ is symmetric. Similarly, $D_s(\cdot, \cdot)$ is symmetric. In conclusion, $\forall f_x\ f_y \in \mathcal{F}$, $d(f_x, f_y) = d(f_y, f_x)$.

Triangle Inequality:

$\forall f_x\ f_y\ f_z \in \mathcal{F}, d(f_x, f_y) \leq d(f_x, f_z) + d(f_z, f_y)$.

Let l_x, l_y, l_z be neural network layers of any width. If $w(l_x) < w(l_y) < w(l_z)$, $d_l(l_x, l_y) = \frac{w(l_y)-w(l_x)}{w(l_y)} = 2 - \frac{w(l_x)+w(l_y)}{w(l_y)} \leq 2 - \frac{w(l_x)+w(l_y)}{w(l_z)} = d_l(l_x, l_z) + d_l(l_z, l_y)$. If $w(l_x) \leq w(l_z) \leq w(l_y)$, $d_l(l_x, l_y) = \frac{w(l_y)-w(l_x)}{w(l_y)} = \frac{w(l_y)-w(l_z)}{w(l_y)} + \frac{w(l_z)-w(l_x)}{w(l_y)} \leq \frac{w(l_y)-w(l_z)}{w(l_y)} + \frac{w(l_z)-w(l_x)}{w(l_z)} = d_l(l_x, l_z) + d_l(l_z, l_y)$. If $w(l_z) \leq w(l_x) \leq w(l_y)$, $d_l(l_x, l_y) = \frac{w(l_y)-w(l_x)}{w(l_y)} = 2 - \frac{w(l_y)}{w(l_y)} - \frac{w(l_x)}{w(l_y)} \leq 2 - \frac{w(l_z)}{w(l_x)} - \frac{w(l_x)}{w(l_y)} \leq 2 - \frac{w(l_z)}{w(l_x)} - \frac{w(l_z)}{w(l_y)} = d_l(l_x, l_z) + d_l(l_z, l_y)$. By the symmetry property of $d_l(\cdot, \cdot)$, the rest of the orders of $w(l_x)$, $w(l_y)$ and $w(l_z)$ also satisfy the triangle inequality. $\therefore \forall l_x\ l_y\ l_z, d_l(l_x, l_y) \leq d_l(l_x, l_z) + d_l(l_z, l_y)$.

$\forall L_a\ L_b\ L_c$, given $\varphi_{l:a\to c}$ and $\varphi_{l:c\to b}$ used to compute $D_l(L_a,\ L_c)$ and $D_l(L_c, L_b)$, we are able to construct $\varphi_{l:a\to b}$ to compute $D_l(L_a, L_b)$ satisfies $D_l(L_a, L_b) \leq D_l(L_a, L_c) + D_l(L_c, L_b)$.

Let $L_{a1} = \{\ l \mid \varphi_{l:a\to c}(l) \neq \varnothing\ \wedge\ \varphi_{l:c\to b}(\varphi_{l:c\to a}(l)) \neq \varnothing\}$. $L_{b1} = \{\ l \mid l = \varphi_{l:c\to b}(\varphi_{l:a\to c}(l')),\ l' \in L_{a1}\}$, $L_{c1} = \{\ l \mid l = \varphi_{l:a\to c}(l') \neq \varnothing,\ l' \in L_{a1}\}$, $L_{a2} = L_a - L_{a1}$,

$L_{b2} = L_b - L_{b1}$, $L_{c2} = L_c - L_{c1}$.

From the definition of $D_l(\cdot,\cdot)$, with the current matching functions $\varphi_{l:a\to c}$ and $\varphi_{l:c\to b}$, $D_l(L_a, L_c) = D_l(L_{a1}, L_{c1}) + D_l\ (L_{a2}\ , L_{c2})$ and $D_l(L_c, L_b) = D_l(L_{c1}, L_{b1}) + D_l(L_{c2}\ , L_{b2})$. First, $\forall l_a \in L_{a1}$ is matched to $l_b = \varphi_{l:c\to b}(\varphi_{l:a\to c}(l_a)) \in L_b$. Since the triangle inequality property of $d_l(\cdot,\cdot)$, $D_l(L_{a1}, L_{b1}) \leq D_l(L_{a1}, L_{c1}) + D_l(L_{c1}, L_{b1})$. Second, the rest of the $l_a \in L_a$ and $l_b \in L_b$ are free to match with each other.

Let $L_{a21} = \{\ l \mid \varphi_{l:a\to c}(l) \neq \varnothing\ \wedge\ \varphi_{l:c\to b}(\varphi_{l:c\to a}(l)) = \varnothing\}$, $L_{b21} = \{\ l \mid l = \varphi_{l:c\to b}(l') \neq \varnothing,\ l' \in L_{c2}\}$, $L_{c21} = \{\ l \mid l = \varphi_{l:a\to c}(l') \neq \varnothing,\ l' \in L_{a2}\}$, $L_{a22} = L_{a2} - L_{a21}$, $L_{b22} = L_{b2} - L_{b21}$, $L_{c22} = L_{c2} - L_{c21}$.

From the definition of $D_l(\cdot,\cdot)$, with the current matching functions $\varphi_{l:a\to c}$ and $\varphi_{l:c\to b}$, $D_l(L_{a2}, L_{c2}) = D_l(L_{a21}, L_{c21}) + D_l\ (L_{a22},\ L_{c22})$ and $D_l(L_{c2}, L_{b2}) = D_l(L_{c22}, L_{b21}) + D_l(L_{c21},\ L_{b22})$. $\because D_l(L_{a22}, L_{c22}) + D_l(L_{c21}, L_{b22}) \geq |L_{a2}|$ and $D_l(L_{a21}, L_{c21}) + D_l(L_{c22}, L_{b21}) \geq |L_{b2}|$ $\therefore$ $D_l(L_{a2}, L_{b2}) \leq |L_{a2}| + |L_{b2}| \leq D_l(L_{a2}, L_{c2}) + D_l(L_{c2}, L_{b2})$. So $D_l(L_a, L_b) \leq D_l(L_a, L_c) + D_l(L_c, L_b)$. Similarly, $D_s(S_a, S_b) \leq D_s(S_a, S_c) + D_s(S_c, S_b)$. Finally, $\forall f_x\ f_y\ f_z \in \mathcal{F}$, $d(f_x, f_y) \leq d(f_x, f_z) + d(f_z, f_y)$.

In conclusion, $d(f_a, f_b)$ is a metric space distance. $\square$