Efficient Neural Architecture Search with Network Morphism

Haifeng Jin
Department of Computer Science and Engineering
Texas A&M University
jin@tamu.edu

Qingquan Song
Department of Computer Science and Engineering
Texas A&M University
song_3134@tamu.edu

Xia Hu
Department of Computer Science and Engineering
Texas A&M University
xiahu@tamu.edu

ABSTRACT

While neural architecture search (NAS) has drawn increasing attention for automatically tuning deep neural networks, 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. However, network morphism based NAS is still computationally expensive due to the inefficient process of selecting the proper morph operation for existing architectures. As we know, Bayesian optimization has been widely used to optimize functions based on a limited number of observations, motivating us to explore the possibility of making use of Bayesian optimization to accelerate the morph operation selection process. 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. With Bayesian optimization to select the network morphism operations, the exploration of the search space is more efficient. Moreover, we carefully wrapped our method into an open-source software, namely Auto-Keras[1] for people without rich machine learning background to use. Intensive experiments on real-world datasets have been done to demonstrate the superior performance of the developed framework over the state-of-the-art baseline methods.

Keywords

Neural Architecture Search; Bayesian Optimization; Network Morphism; Gaussian Process; Kernel Methods

1. INTRODUCTION

Neural architecture search (NAS), which aims to search for the best neural network architecture given a learning task, has become an effective computational tool in automated machine learning (AutoML). Unfortunately, existing NAS algorithms are usually computationally expensive. The time complexity of NAS could be roughly computed 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 [33, 1, 32, 21] and evolutionary algorithms [23, 7, 18, 25, 29, 22], 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 been successfully applied for neural architecture search [4, 8]. Network morphism is a technique to morph the architecture of a neural network but keep its functionality [5, 27]. 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 [4] 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 [8] 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.

Bayesian optimization has been widely adopted for finding the optimum value of a function based on a limited number of observations. It is usually used to find the optimum point of a black-box function, whose observations are expensive to obtain. For example, it has been used in hyperparameter tuning for machine learning models [26, 14, 9, 11], each observation of which involves the training and testing of a machine learning model, which is very similar to the NAS problem. The unique properties of Bayesian optimization 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. The traditional Newton-like or gradient-based methods cannot be simply applied. Third, the network morphism operations changing one layer in the neural architecture may invoke many changes to other layers to maintain the input and output consistency, which is not defined in previous work. The network morphism operations are complicated in a search space of neural architectures with skip-connections.

In this paper, an efficient neural architecture search with network morphism is proposed, which utilizes Bayesian optimization to

[1]The code is available at http://autokeras.com.

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 is designed specially for the tree-structure search space to enable Bayesian optimization to select from the operations. The optimization methods can balance between the exploration and exploitation during the optimization. In addition, a network-level morphism is defined to address the complicated changes in the neural architectures based on previous layer-level network morphism. Our method is wrapped into an open-source software, namely Auto-Keras. The proposed approach is evaluated on benchmark datasets and compared with the state-of-the-art baseline methods. The main contributions of this paper are summarized as follows:

- An efficient neural architecture search algorithm with network morphism is proposed.
- Bayesian optimization for NAS with neural network kernel, tree-structured acquisition function optimization, and a network-level morphism is proposed.
- An open-source software, namely Auto-Keras, is developed based on our method for neural architecture search.
- Intensive experiments are conducted on benchmark datasets to evaluate the proposed method.

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 find 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. GUIDE 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. Equipped with the view 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 performance. There are two main challenges in designing the method for morphing the neural architectures with Bayesian optimization. It highly reduces the desired number of trained neural architectures and avoids the merely neighborhood-wise exploration. Next, we introduce three key components separately in the subsequent sections coping with the three design challenges. The time complexity of update and generation is low enough comparing to the observation.

3.1 Neural Network Kernel

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. It is impractical to vectorize every neural architecture due to the uncertainly large 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 [30], we propose an edit-distance kernel for neural networks, which consistent with our idea of using 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:

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

where function $d(\cdot, \cdot)$ denotes the edit-distance of two neural networks, whose range is $[0, +\infty)$, ρ is the Bourgain algorithm [2], which distorts the distance to ensure the validity of the kernel.

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 [31]. 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 denote 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, $L_a = \{l_a^{(1)}, l_a^{(2)}, \ldots\}$ and $L_b = \{l_b^{(1)}, l_b^{(2)}, \ldots\}$ 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, $S_a = \{s_a^{(1)}, s_a^{(2)}, \ldots\}$ and $S_b = \{s_b^{(1)}, s_b^{(2)}, \ldots\}$ 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

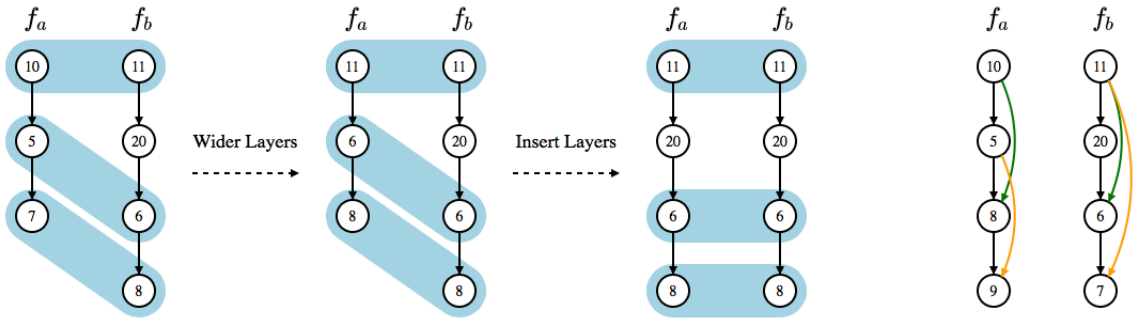


Figure 1: Neural network kernel

widening a layer into another defined in Equation (5), where $w(l)$ is the width of layer l.

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

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 numbers on the nodes are the widths of the intermediate tensors. The matchings between the nodes are marked by light blue. The nodes are intermediate tensors output by the previous layers, which 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). So a matching between the nodes can be seen as a matching between the layers. To morph f_a to f_b, 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:

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

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

Calculating D_s: The intuition of the D_s term is to measure the edit-distance of matching the most similar 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:

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

where we assume $|S_a| < |S_b|$. $(|S_b| - |S_a|)$ measures the total edit-distance for not matched skip-connections. Each of the not mapped skip-connections in S_b means a new skip connection that needs to be inserted in f_a. The mapping function $\varphi_s : S_a \to S_b$ is an injective function. The edit-distance for two matched skip-connections is $d_s(s_a^{(i)}, s_b^{(j)})$ defined as:

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

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 parts of the graph, each skip-connection is a node in its corresponding part. 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) [16].

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.

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 $w(l_z) \leq w(l_x) \leq w(l_y)$, $d_l(l_x, l_y) = \frac{w(l_y)-w(l_x)}{w(l_y)} = 2 - \frac{w(l_y)}{w(l_y)} - \frac{w(l_x)}{w(l_y)} \leq 2 - \frac{w(l_z)}{w(l_x)} - \frac{w(l_z)}{w(l_y)} \leq 2 - \frac{w(l_z)}{w(l_x)} - \frac{w(l_z)}{w(l_y)} = d_l(l_x, l_z) + d_l(l_z, l_y)$. 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. □

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

Proof of Theorem 2: The network edit-distance $d(\cdot, \cdot)$ is a distance in metric space, the proof of which is in Theorem 1 in the Appendix. The Bourgain algorithm [2] denoted as $\rho(\cdot)$ in Equation (2) preserves the symmetry and definiteness property of $d(f_a, f_b)$. Therefore, $\forall f_x f_y \in \mathcal{F}$, $\kappa(f_x, f_y) = \kappa(f_y, f_x)$ and $\kappa(f_x, f_y) = 0 \iff f_x = f_y$. The kernel matrix of generalized RBF kernel in the form of $e^{-\gamma d(x,y)}$ is positive definite if and only if there is an isometric embedding in Euclidean space for the metric space with metric d [10]. Any finite metric space distance can be isometrically embedded into Euclidean space by changing the scale of the distance measurement [19]. So Bourgain algorithm $\rho(\cdot)$ distort $d(\cdot, \cdot)$ to be isometrically embeddable in Euclidean space, Therefore, the kernel matrix is always positive definite. So $\kappa(f_a, f_b)$ is a valid kernel. □

3.2 Acquisition Function

The second challenge we need to address is acquisition function optimization. The traditional acquisition functions are defined on Euclidean space, the methods for optimizing which are not applicable to the tree-structured search via network morphism. A novel method to optimize the acquisition function is proposed for tree-structured space.

Algorithm 1: Optimize Acquisition Function

Input: $\mathcal{H}, r, T_{low}$
Output: f, O

$T \leftarrow 1, Q \leftarrow$ `PriorityQueue()`, $c_{min} \leftarrow$ lowest c in $\mathcal{H}$
foreach $(f, \boldsymbol{\theta_f}, c) \in \mathcal{H}$ **do**
  Q.`push(`f`)`
while $Q \neq \varnothing$ **and** $T > T_{low}$ **do**
  $T \leftarrow T \times r$, $f \leftarrow Q$.`pop()`
  foreach $o \in \Omega(f)$ **do**
    $f' \leftarrow \mathcal{M}(f, \{o\})$
    if $e^{\frac{c_{min}-\alpha(f')}{T}} >$ `rand()` **and** f' is not duplicate **then**
      Q.`push(`f'`)`
    if $c_{min} > \alpha(f')$ **then**
      $c_{min} \leftarrow \alpha(f'), f_{min} \leftarrow f'$
return The nearest ancestor of f_{min} in $\mathcal{H}$, the operation sequence to reach f_{min}

Upper-confidence bound (UCB) in Equation (9) is chosen as our acquisition function.

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

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 properties that 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 $\mathcal{H} = \{(f^{(i)}, \boldsymbol{\theta}^{(i)}, c^{(i)})\}$, which is a property to be used in our algorithm. 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 minimization 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 state-of-the-art acquisition function maximization techniques, *e.g.*, gradient-based or Newton-like method, are designed for numerical data, which do not apply in the tree-structure space. TreeBO [12] has proposed a way to maximize the acquisition function in a tree-structured parameter space. Only its leaf nodes have acquisition function values, which is different from our case. Moreover, the proposed solution is surrogate multivariate Bayesian optimization model. In NASBOT [13], they use an evolutionary algorithm to optimize the acquisition function. They are both very expansive in computing time. To minimize our acquisition function, we need a method to efficiently minimize the acquisition function in the tree-structured space.

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

As shown in Algorithm 1, the algorithm takes minimum temper-

ature 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 and a sequence of operations O. From line 2 to line 3, the searched architectures are pushed into the priority queue, in which they are sorted according to their 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 $c^{(i)}$. $s[f]$ records which history architecture is f morphed from. $O[f]$ records what operations are applied to morph $s[f]$ to f. From line 4 to line 11 is the loop minimizing 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 5 to 6, where $\Omega(f)$ is all the possible operations to morph the architecture f, $\mathcal{M}(f, o)$ is the function for 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 8, where $e^{\frac{c_{min}-\alpha(f')}{T}}$ is a typical acceptance function in simulated annealing. Notably, on line 10, the cost value is directly compared with the acquisition function value given the property UCB.

3.3 Network Morphism

The third challenge is to maintain the input and 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 network-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 network-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 $f \in \mathcal{F}$ [8], which can all be reflected in the change of the computational graph. 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 intermediate output tensor to be widened. Widen here could be making the output vector of a fully-connected layer longer, or adding more filters to the previous convolutional layer of u. 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 [5]. 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. Third, the effect area of $concat(G, u_0, v_0)$ can be similarly defined by the following rules: 1. $u_0 \in \gamma$. 2. $v_0 \in \gamma$. 3. $v \in \gamma$, if $\exists e_{u\to v} \notin L_s$, $u \in \gamma$. 4. $v \in \gamma$, if $\exists e_{v\to u} \notin L_s$, $u \in \gamma \wedge u \neq u_0 \wedge u \neq v_0$. The L_p and L_n is the same as defined in the wide operation. Additional pooling layers are also needed for the skip-connection.

3.4 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. However, the efficiency of the update and the generation is also important, since they must not become the bottleneck. 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

An open-source software, namely Auto-Keras, is developed using on our method for neural architecture search, in which Keras [6] is used for the construction and training of the neural networks. Similar to SMAC [11], Auto-WEKA [26], and Auto-Sklearn [9], the goal is to enable domain experts who is not familiar with machine learning technologies to use deep neural networks easily. 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. 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.

4.1 Components

The Auto-Keras package consists of four major components, which are Classifier, Searcher, Graph, and Trainer. The Classifier is the program interface class, which is responsible for calling corresponding modules to complete certain tasks. The Searcher is the module containing Bayesian optimization. Each time its search function is called, it would run one round of Bayesian optimization, which consists of update, generation, observation. The Graph is the class of computational graph of neural networks, which has member functions implemented for the network morphism operations. It is called by the Searcher to morph the neural architectures. The Trainer is the class to train a given neural network with the training data in a separate process to avoid the GPU memory leak. It is capable of various training techniques to improve the final performance of the neural network including data augmentation and automated detection of convergence.

4.2 Interface

The design of the application programming (API) interface follows the classic design of the Scikit-Learn API [20, 3]. 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 other similar packages is it takes much longer, since it needs to train a number of deep neural networks. 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, all the searched architectures are visualized in the saved directory as PNG files. Third, the user can export the search results, which are neural architectures, as saved Keras models for other usages. Fourth, 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 Convergence

To fully automate the entire process from input data to the final trained neural network, automated detection of convergence is needed both during the search and the final training of the found best architecture. We use the same strategy as the early stop strategy in the multi-layer perceptron algorithm in Scikit-Learn [20]. It sets a maximum threshold τ. If the loss of the validation set doesn't decrease in τ epochs, the training stops. Since different architectures may require different numbers of training epochs, comparing with the many state-of-the-art methods using a fixed number of training epochs, the convergence detection strategy is more adaptive to different architectures. It would better ensure the correlation between the performance of a certain neural architecture during the search and its final performance when fully trained, which is essential for the entire neural architecture search process.

4.4 Parallelism

The program can run across multiple GPUs and CPUs at the same time. It relies on the inner parallel mechanism of Keras to run across multiple GPUs during the training of the neural networks. The functional programming paradigm in python enables the rest of the computation to run in parallel across multiple CPUs. However, if we do the observation, 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 update and generation in separated processes. A training queue is maintained as a buffer. In each Bayesian optimization cycle, the Trainer takes one architecture from the queue and trains it. The Searcher runs the generation in parallel to search the next architecture to train. After observation, the model is updated with the architecture and the observed performance. After generation, the newly generated architecture is pushed into the queue. In this way, the idle time of GPU and CPU are dramatically reduced to improve the efficiency of the search process.

5. EXPERIMENTS

In the experiments, we aim at answering the following questions. 1) What is the effectiveness of the search algorithm with limited running time? 2) What are the influences of the important hyperparameters of the search algorithm? 3) Does the edit-distance neural network kernel correctly predict the similarity in actual performance?

Table 1: Classification error rate

Methods	MNIST	CIFAR10	FASHION
RANDOM	0.013	0.420	0.092
GRID	0.016	0.274	0.129
MCMC	**0.012**	0.315	0.134
SMAC	0.026	0.337	0.443
SEAS	0.013	0.197	0.080
NASBOT	NA	0.124	NA
BFS	0.035	0.285	0.095
BO	0.026	0.307	0.090
NASNM	**0.012**	**0.123**	**0.078**

Datasets Three benchmark datasets, MNIST [17], CIFAR10 [15], and FASHION [28] are used for the experiments. They require very different neural architectures to achieve good performance.

Baselines Four categories of baseline methods are used to compare with our work. two straightforward solutions, random search (RAND) and grid search (GRID), two traditional baselines, MCMC [24] and SMAC [11], two state-of-the-art neural architecture search work: SEAS [8], NASBOT [13], and two variants of our proposed methods, BFS and BO. MCMC and SMAC tunes the 16 hyperparameters of a three-layer convolutional neural network, including the width, dropout rate, and regularization rate of each layer. 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 published in the paper. The architecture is implemented and trained on the dataset originally used for the search. The BFS methods is a variant of our own method, which replace the Bayesian optimization with the breadth-first search. BO is another variant, which does not use network morphism for acceleration. Finally, our proposed method is NASNM.

5.1 Evaluation of Effectiveness

The experiments of evaluating the effectiveness of the proposed method are conducted as follows. First, each dataset is split by 60-20-20 into training, validation and testing set. Second, run the method for 12 hours on a single GPU (NVIDIA GeForce GTX 1080 Ti) on the training and validation set. Third, the output architecture is trained with both training and validation set. Fourth, 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 model trainer is used to train the neural networks for 200 epochs for all the experiments, which contains several techniques to enhance the performance, including, layer regularization, data augmentation, learning rate control, and etc.

The results are shown in Table 1. Our method achieved lowest error rate on all of the datasets. MNIST requires simple neural architectures to achieve lower error rate. Complicated neural architectures are likely to overfit the dataset. Shown in the results, our method is able to avoid the overfitting issue during the search. Similarly, CIFAR10 is a dataset require more complicated neural architectures and is likely to be underfitted. FASHION is an intermediate dataset which could both be underfitted and overfitted. On these two datasets, our method also achieved the lowest error rate compared with the baselines. Most of the simple and traditional approaches performed well on the MNIST dataset, but not very well on the CIFAR10 dataset. For simple approaches like random search and grid search, they don't work on CIFAR10 since they can only try a limited number of architectures blindly. For traditional approaches, the main reason is their inability to change the depth

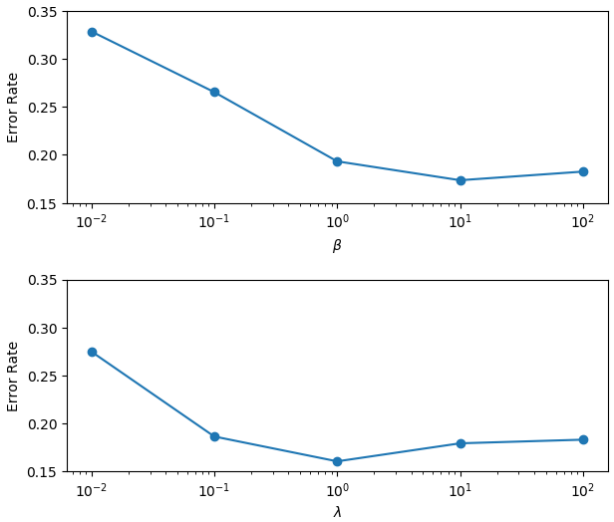


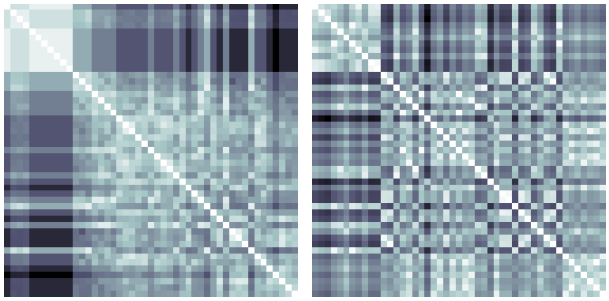
Figure 2: Parameters

and skip-connections of the architectures. SEAS [8] also performed well on all three datasets. The error rate of CIFAR10 is a little higher. It is because the good architectures are farther from the initial architecture in terms of network morphism operations, and the hill-climbing strategy only takes one step at a time in morphing the current best architecture. NASBOT [13] performed well on CIFAR10. It also uses Bayesian optimization, but it is not a network morphism based method. The training of the neural networks takes longer. The low error rate was achieved by parallel searching on multiple GPUs. BFS has the similar problem as hill-climbing, it always searches a large number of neighbors first, which make it not likely to reach the good results far from the initial architecture. BO can jump far from the initial architecture. But without network morphism, it needs to train each neural architecture for much longer, which limits the number of architectures it can search within a given time. Some results may not be as good as in some papers. The main reason is all the methods use the default training settings, including data preprocessors, optimizers, batch size and etc, to eliminate the influence of unwanted factors.

5.2 Parameter Sensitivity Analysis

There are several hyperparameters in our proposed method, λ in Equation (3) and β in Equation (9), r and T_{low} in Algorithm 1. Since r and T_{low} are just normal hyperparameters of simulated annealing, the experiments focus on λ and β. λ balances between the distance of layers and skip connections in the kernel function. β is the weight of the variance in the acquisition function, which balances the exploration and exploitation of the search strategy. The setup for the parameter experiments is similar to the performance experiments, except for the final training in step three.

As shown in the top part of Figure 2, with the increase of β from 10^{-2} to 10^{2}, the error rate decreased and increased. If the β is too low, the search process is not explorative enough to search the architectures far from the initial architecture. If it is too high, the search process would keep exploring the far points instead of trying the most promising architectures. As shown in the bottom part of Figure 2, λ influences the performance similar to β. If λ is too low, the differences in the skip-connections of two neural architectures are ignored. If it is too high, 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 to achieve a good performance of the entire framework.



(a) Kernel matrix (b) Performance matrix

Figure 3: Kernel and performance matrix visualization

5.3 Evaluation of Kenrel Quality

To show the quality of the edit-distance neural network kernel, we investigate the difference between the two matrices K and 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 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. Here we use CIFAR10 as the dataset, and error rate as the cost metric. Since the values in K and P are in different scales, both matrices are normalized to the range -1 to 1.

$\boldsymbol{K}$, $\boldsymbol{P}$, and $\boldsymbol{K} - \boldsymbol{P}$ are visualized in Figure 3a and 3b. Lighter color means a larger value. There are several patterns shown in the figures. First, the white diagonal of Figure 3a and 3b. Since the definiteness of the kernel, $\kappa(f_x, f_x) = 1, \forall f_x \in \mathcal{F}$, 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 square on the upper left of Figure 3a. These are the initial neural architectures to train the Bayesian optimizer, which are neighbors to each other in terms of network morphism operations. The similar pattern in Figure 3b indicates that, when the kernel measure two architectures as similar, they tend to have similar performance. Third, the dark region on the top and left of Figure 3a. The main reason is the rest of the architectures are dissimilar to the initial architectures. The similar pattern in Figure 3b shows that, when kernel measure two architectures as dissimilar, they tend to have different performance. The dark color on the upper right corner of Figure 3a shows a small flaw of the kernel, that the quantity of the difference in the performance is not accurately measured. Fourth, the kernel matrix is smoother than the performance matrix, which is because there is noise in the measured performance due to various training issues. Finally, we quantitatively measure the difference between $\boldsymbol{K}$ and $\boldsymbol{P}$ with mean square error (MSE). The range of the values is (-1, 1). The MSE is 1.12×10^{-1}.

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 software, 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.

7. REFERENCES

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

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

[3] 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* (2013), pp. 108–122.

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

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

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

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

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

[9] 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* (2015), pp. 2962–2970.

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

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

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

[13] Kandasamy, K., Neiswanger, W., Schneider, J., Poczos, B., and Xing, E. Neural architecture search with bayesian optimisation and optimal transport. *arXiv preprint arXiv:1802.07191* (2018).

[14] 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* (2016), 1–5.

[15] Krizhevsky, A., and Hinton, G. Learning multiple layers of features from tiny images.

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

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

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

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

[20] 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* (2011), 2825–2830.

[21] 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).

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

[23] 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).

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

[25] 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* (2017), ACM, pp. 497–504.

[26] 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* (2013), ACM, pp. 847–855.

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

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

[29] Xie, L., and Yuille, A. Genetic cnn. *arXiv preprint arXiv:1703.01513* (2017).

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

[31] 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 (2009), 25–36.

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

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