EFFICIENT NEURAL ARCHITECTURE SEARCH FOR AUTOMATED DEEP LEARNING

A Dissertation

by

HAIFENG JIN

Submitted to the Office of Graduate and Professional Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Chair of Committee,	Xia Hu
Committee Members,	James Caverlee
	Yang Shen
	Zhangyang Wang
Head of Department,	Scott Schaefer

May 2021

Major Subject: Computer Science

ABSTRACT

Deep learning has been widely applied for its success in many real-world applications. To adopt deep learning, people often need to go through a non-trivial learning curve like learning the foundation of machine learning theory and how to use the deep learning libraries. Automated deep learning has emerged as an important research topic to reduce the prerequisites for adopting deep learning. Neural architecture search (NAS), as the most important component of the automated deep learning process, is to solve the problem of automatically finding a good neural architecture. However, existing NAS methods suffer from several problems. It usually has a high requirement for computational resources and cannot be efficiently and jointly tuned with other parts of the deep learning solution like the preprocessing steps or the optimizer hyperparameters. This dissertation aims to improve the efficiency of NAS as a stand-alone process and as an important step in the overall automated deep learning process. We propose a series of methods and frameworks for extracting information from the neural architectures, improving the search and evaluation efficiency of NAS, enabling joint tuning with other hyperparameters, and automatically selecting data augmentation strategies.

ACKNOWLEDGMENTS

There are many who helped me during my Ph.D. study. I would like to take a moment to thank them. I wish to thank my advisor, Dr. Xia Hu, all my committee members, Dr. James Caverlee, Dr. Yang Shen, Dr. Zhangyang Wang, and my labmates and friends for their advice and support.

CONTRIBUTORS AND FUNDING SOURCES

Contributors

This work was supported by a dissertation committee consisting of Professor Xia Hu [advisor], Professor James Caverlee of the Department of Computer Science and Engineering at Texas A&M University, and Professor Yang Shen of the of the Department of Electrical and Computer Engineering at Texas A&M University, and Professor Zhangyang Wang of the Department of Electrical and Computer Engineering at the University of Texas at Austin.

All the work conducted for the dissertation was completed by the student independently.

Funding Sources

This work is, in part, supported by DARPA (#FA8750-17-2-0116) and NSF (#IIS-1718840 and #IIS-1750074). The views, opinions, and/or findings expressed are those of the author(s) and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

TABLE OF CONTENTS

Page

LIST OF FIGURES

FIGURE Page

LIST OF TABLES

TABLE Page

1. INTRODUCTION

Machine learning has been widely applied in various real-world problems [2]. However, applying machine learning to real-world problems is a complicated process. For individuals, they need to have rich expertise and experiences in both machine learning and programming [3]. For businesses, they can either buy the machine learning services from a third party or hire machine learning experts and software engineers to do the job. Such a huge barrier prevents domain experts in various fields and small businesses, who own valuable data, to use machine learning to build useful applications [4]. To reduce the barrier for people to apply machine learning, automated machine learning (AutoML) has emerged as a research problem.

1.1 Background

In this section, the background knowledge of the dissertation is introduced, including the basics of AutoML, AutoML for deep learning, and the basics of neural architecture search.

1.1.1 Automated Machine Learning (AutoML)

AutoML aims at automating the process of applying machine learning to real-world problems [5] by providing users with end-to-end machine learning solutions. The process may include preprocessing, feature engineering, model selection, hyperparameter tuning, and others.

On the current stage, there are mainly three goals of AutoML, which can be summarized as follows. (1) **Usability**: AutoML aims to provide machine learning as an available tool for people with limited computer science and machine learning background. The domain experts with valuable data and knowledge can easily develop their machine learning models and applications with AutoML. (2) **Productivity**: AutoML aims to increase the productivity of machine learning engineers. Machine learning engineers nowadays usually spend a long time trying out different models and tuning the hyperparameters of the models. With the assistance of AutoML, they can automate the repetitive work of tuning the hyperparameters. Moreover, they can inject their expertise for selecting and designing the models into the AutoML process to accelerate the overall process

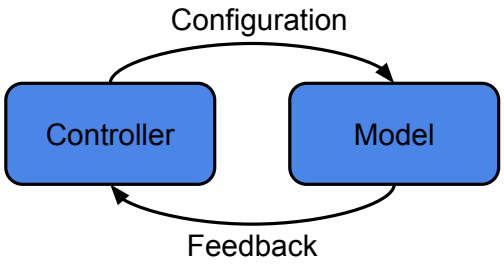


Figure 1.1: Process of AutoML

even more. (3) **Performance**: AutoML aims to find better machine learning models than the models manually designed by human experts. With the strong computing power of large clusters with GPUs and TPUs [6], AutoML can explore a massive amount of different models, among which there may be some models that are even better than the state-of-the-art models designed by humans experts today.

The process of AutoML algorithms are as shown in Figure 1.1. The controller generates a new configuration, which can be instantiated into a model. The model's performance is evaluated and feedback to the controller. The controller learns from the feedback and tries to generate a new configuration. Through this loop, the controller can gradually learn the good and bad configurations so that it can generate configurations with higher performances. The configuration can be the values of the hyperparameters or the operations for feature engineering, and so on. The model is not limited to machine learning models like support vector machines or neural networks. It can also be an end-to-end machine learning pipeline. For example, the model may contain three steps, which are data normalization, feature selection, and support vector machine for classification.

There are several important concepts involved in the process of AutoML: hyperparameters, search space, search algorithm, and evaluation method. Hyperparameters are the parameters, which cannot be learned from the data, whose value is assigned in advance of the learning process. For example, the learning rate of a linear regression algorithm. The search space is the set of all possible value combinations of the hyperparameters, which is the set of all the possible configurations that can be generated by the controller. The search algorithm is the algorithm used by the controller

to learn from the configurations and the performances of the models. There are several famous approaches for the search algorithm, including bandit approaches [7]. Bayesian optimization [8, 9]. The evaluation method is the method used to evaluate the performance of a model. Cross-validation is widely used for model evaluation. Many other techniques are also used for accelerating the evaluation process [10].

1.1.2 AutoML for Deep Learning

Deep learning has evolved fast in recent years with applications in various fields including computer vision [11], and natural language processing [12]. To automate the process of applying deep learning to real-world problems, AutoML for deep learning, also known as automated deep learning (AutoDL), has become an important problem.

AutoDL is a subfield of AutoML. The process of AutoDL can also be described by Figure 1.1. The only difference is the model is limited to deep neural networks and their related processing and training steps. It automates the design and training process of deep neural networks and the related preprocessing and postprocessing steps.

1.1.3 Neural Architecture Search

Within the scope of AutoDL, there is a topic, which has drawn increasing attention recently, named neural architecture search. It further narrows down the definition of the model in Figure 1.1 to deep neural networks only without the training or data processing steps. Neural architecture search aims to automatically design the best neural architecture for a given task. Neural architecture is the configuration of a neural network, which can include deciding the number of layers in the neural network, the number of neurons each layer contains, whether to use skip connections between the layers, and so on.

1.2 Motivation

The biggest barrier that prevents neural architecture from being widely used is low efficiency. There are several challenges to be tackled to achieve efficient neural architecture search, which are introduced as follows.

The first challenge to be tackled is that the controller needs to efficiently learn from the neural architectures. It needs to learn the relation between the neural architectures and their performances on the target dataset. The search space of neural architecture search is different from traditional AutoML, which consists of a set of different types of hyperparameters, for example, learning rate, kernel type. Neural architecture search space is a cleaner search space that consists of only computational graphs. Therefore, a method is needed to efficiently learn from the graph data to extract useful information from the neural architectures. It motivates us to research the graph representation learning methods to solve this problem.

The second challenge to be tackled is the efficiency of neural architecture search. It usually completely trains a neural network to evaluate performance as the feedback to the controller. Since the NAS needs to evaluate a large number of neural architectures, the total computational cost is huge. It motivates us to explore a more efficient search algorithm in the controller and a more efficient way to train and evaluate the neural architectures during the search.

Third, there are many other hyperparameters besides the neural architecture to be tuned. For example, to apply deep learning to a real-world problem, one needs to decide the preprocessing steps, the optimizer to use to train the neural network, and the learning rate. Moreover, these hyperparameters are not independent of the neural architectures. Different neural architectures may need different learning rates and optimizer to achieve their best performances. It motivates us to explore how to tune the hyperparameters and the neural architectures jointly and efficiently.

The fourth challenge to be tackled is that data augmentation is inefficient to automate. Data augmentation is critical to the performance of neural network training [13]. It contains many hyperparameters, which enlarges the search space exponentially, which increases the burden of the search algorithm while exploring the search space. Moreover, it often requires training the neural network entirely for each of the evaluations when being automated. It motivates us to separate the automation of the data augmentation from the hyperparameter tuning process and propose a more efficient way to automatically find good data augmentation strategies.

1.3 Contributions

This dissertation addresses four primary problems towards efficient neural architecture search, which regards different parts of the overall AutoML framework ranging from search algorithm, search space, and evaluation method. (1) How to efficiently and effectively learn vector representations from graphs, which is the basic form of the neural architectures? (2) How to efficiently search and evaluate neural architectures for a given dataset and task? (3) How to efficiently tune the hyperparameters together with the neural architectures for deep learning tasks? (4) How to efficiently select a good data augmentation strategy for a given dataset and task?

We propose a series of methods to answer these questions. The key contributions of this dissertation can be summarized as follows.

- We propose a novel graph representation learning method, which can efficiently and effectively learn a smooth vector representation with an autoencoder architecture to leverage the discriminative information in graphs based on class labels.

- We propose a novel efficient neural architecture search algorithm based on network morphism guided by Bayesian optimization. It enables Bayesian optimization, which is the most widely used approach to traditional hyperparameter tuning problems, in the neural architecture search space. Also, it warm-starts the new neural architecture with weights in previously trained neural networks.

- We propose a novel framework for hyperparameters and neural architectures joint tuning and a corresponding greedy search algorithm. It maps the neural architecture and the rest of the hyperparameters into the same hyperparameter space. It also proposes a search algorithm to more efficiently explore the search space.

- We propose a novel measure for quantifying the diversity of the augmented data and use it to automatically and dynamically select the data augmentation strategies for each epoch during the training. The method does not use a search loop to significantly reduce the search time of

automated data augmentation.

1.4 Dissertation Overview

The remainder of this dissertation is organized as follows:

- **Section 2-5**: We introduce each of the proposed methods in the contributions above in details. We conducted experiments to validate the effectiveness and efficiency of the proposed method. The results are shown in each of the chapters.

- **Section 6** We conclude the dissertation by summarizing the contributions and propose potential future work to extend our research.

2. GRAPH REPRESENTATION LEARNING*

In neural architecture search, the controller needs to learn the relation between the neural architectures and their performance. However, the neural architecture is not a ready-to-use form for many search algorithms, which can be potentially used in the controller. The search algorithms may only work in Euclidean space and only accept vectorized data. Therefore, learning vector representations for neural architectures is an effective way to enable a wider range of search algorithms in the controller.

Neural architectures are computational graphs, which are attributed graphs with multiple attributes on the edges and the nodes. In a computational graph of a neural network, each node is an intermediate output tensor, whose attributes can be its shape, while each edge is a layer, whose attributes can be the number of neurons it contains. Therefore, the problem of learning vector representations for neural architectures is mapped to a graph representation learning problem. A novel solution is proposed in this chapter to solve the problem.

2.1 Introduction

Besides neural architectures, graphs are widely used to represent macrostructures of relational instances, such as airline networks, publication connections, and social communications [14]. Beyond the single graph setting, multiple microstructures are also ubiquitous in various real-world data such as protein graphs, molecular expressions, and control flows. In such cases, each data instance is a graph instead of a node resulting in higher complexity and difficulty in analysis and applications. Graph representation learning aims at deriving informative low-dimensional representations to prepare graphs for a variety of graph mining tasks such as graph classification [15] and clustering [16].

A widely used approach to graph representation learning is the graph kernel, which measures the similarity between graphs with vector inner product [17]. The key idea of graph kernels is to

*Reprinted with permission from “Discriminative Graph Autoencoder" by Haifeng Jin, Qingquan Song, Xia Hu, 2018. IEEE International Conference on Big Knowledge (ICBK), pp. 192-199, Copyright 2018 by IEEE.

predefine a set of representative substructures and count their frequency in the graphs, known as frequency-based methods.

Graph kernels extend kernel-based methods on graphs. Most of the work maps graphs to vector space by counting the presence of specific substructures, which could be considered as frequency-based methods.

They use the presence of a certain set of sub-structures as the features of a graph. There are three main approaches for the state-of-art methods, namely limited-sized subgraphs [18, 19, 20, 21, 22, 23, 24], graph kernels based on subtree patterns [17, 25], and graph kernels based on walks and paths [26, 27, 28].

Some works improve traditional graph kernels and their performance on downstream tasks, such as solving the problem of the dominance of certain dimensions of the learned vector representations [29], find discriminative subgraphs as substructures to learn better vector representations for classification tasks [30, 31]. Besides, Kong et al. [32] try to reduce the labeling cost for graph data for classification tasks. Saigo et al. [33] proposed a method to collect informative patterns progressively through mathematical programming.

Inevitably, these solutions all suffer from the following problems. First, the complexity is high. For the subgraph based methods, it is extremely expansive even to calculate the number of occurrences of a subgraph in a given graph. For other methods, the complexity is at least $O(\sum v_i)$, which is the sum of all the values in the vector representation, where the learned vector representation is $V = [v_1, v_2, \ldots, v_n]$. Second, the vector representation is sparse. Even two similar graphs may not share many non-zero dimensions in their vector representations, which significantly degraded the performance of the downstream tasks. Third, the values in the vector representations are discrete, which would result in a nonsmooth vector space. The vector representations would be rigid and lose more information in the original graph. The effects of such rigidity are shown in the experiments.

Besides the traditional frequency-based approaches, some recent work aims to solve the problems above. Yanardag et al. [34] proposed a method to measure the similarity between the substructures

selected. It is more accurate in measuring the similarity values. Based on this work, Narayanan et al. [14] proposed a new way to learn vector presentations for subgraphs to better solve the sparsity problem. In addition, some other state-of-the-art work [35, 36, 37] solve the sparsity and discrete problem by changing the kernel function. However, as we mentioned before, they can solve the sparsity problem but cannot produce vector representation. It is compatible with kernel-based methods like SVM, but not compatible with other methods that require vector representations of the input, such as neural networks.

Deep learning has been widely applied to feature extraction. Work has been done on network data [38], image data and text data[39]. The embedding and feature-learning techniques all try to map one form of data into the latent space, so that every data instance is represented in a vector representation preserving its original properties, which is much easier for further learning or processing than the original form of the data. Instead of text, image, or a node in a network as a data instance, we have graphs as data pieces. However, only a few works focus on graph representations.

Duvenaud et al. [40] also proposed a method for running convolutional neural networks on molecular data in graph form. It generalizes standard molecular feature extraction methods based on circular fingerprints. Scarselli et al. [41] proposed a graph neural network that used recurrent neural networks for graph data. It maps the nodes of a graph and one of its nodes to the Euclidean space which can process various types of graphs. It is useful for rooted graphs since a root node has to be selected for the learning process. Li et al. [42] modified the graph neural network to use gated recurrent units and modern optimization techniques. These two works only did unsupervised learning which did not take the valuable labeling data into consideration.

The modern convolutional neural networks and recurrent neural networks are also involved in the graph similarity measurement problem. Niepert et al. [43] proposed a method for preprocessing the graph data to be in the input format of a convolutional neural network. Their emphasis is placed on fast formatting graphs into suitable inputs of the neural network, which is similar to our graph sampling method. It is used as the baseline method for efficiency evaluation in the experiment part. More details are introduced in the experiment section.

Autoencoder has been successfully applied to many real-world applications such as image representations learning [44], machine translation [12] and video representation learning [45]. The success of autoencoder motivates us to explore its nice properties which could be a better fit for our problem. First, the values in the learned representations are continuous. The information is more fine-grained than that in the raw integer representations. Second, the produced representation length is flexible. The user could define a proper length of the learned representation to avoid the sparsity problem [46]. Third, it is efficient by avoiding the complicated feature engineering process. In this chapter, we propose to investigate whether a novel computational framework based on autoencoder could better tackle the challenges in learning graph representations.

This is a nontrivial task to design an autoencoder for learning graph representations because of the following reasons. First, unlike image or video data, graphs are not readily prepared for an autoencoder which requires a vector input, while graphs are structural data. Second, graphs are of various sizes. The architecture of the autoencoder needs to be modified to take different sizes of input. Third, graphs have hierarchical information. For example, computer programs have subroutines as subgraphs; and chemical compounds have functional groups as subgraphs. To take full advantage of hierarchical information, the autoencoder needs to consider both overview and details of the graphs.

In addition, the discriminative information in the graphs could be important. Graph data usually have valuable class label information with them. For example, computer programs may be labeled as malicious or benign; and chemical compounds may be labeled as acid or alkaline. The labels make the representations more meaningful for the downstream applications, e.g., make classification more accurate or make the visualization more illustrative. While discriminative information could be useful in learning graph representations, it is simply ignored in much related work [30]. Thus we also propose to study how the discriminative information could be naturally embedded in the learned graph representation.

To tackle the above challenges, in this chapter, we study the problem of learning graph representations. We aim at answering the following questions. (1) How to efficiently and effectively learn

vector representations from graphs? (2) How to incorporate the discriminative information in the graphs into the learned representation based on class labels? By investigating these questions, a novel method to learning graph vector representations is proposed, namely Discriminative Graph Autoencoder (DGA). We summarize the main contributions of this chapter as follows:

- A novel graph representation learning method DGA is proposed, which is able to efficiently and effectively learn a smooth vector representation.
- Present an autoencoder architecture to leverage the discriminative information in graphs based on class labels.
- Validate DGA effectiveness and efficiency through classification and visualization on real-world datasets.

2.2 Problem Statement

Notations and the mathematical definition of the core problem to solve are presented as follows.

2.2.1 Notations

Given a set of graphs $\mathbb{G} = \{G_1, ..., G_n\}$ where n is the number of graphs, each graph is denoted as $G = (V, E) \in \mathbb{G}$, where $V = \{v_1, ..., v_{|V|}\}$ denotes its vertex set and $E = \{e_{uv} | \text{if an edge connecting } u \text{ and } v \text{ exists}\}$ is its edge set. $l(e)$ and $l(v)$ denote the labels of edge e and vertex v, respectively. $p(v)$ denotes the index of v in the canonical permutation of vertices in G. $Y = \{y_1, y_2, \ldots, y_n\}$ denotes the graph level label of G. A_G is the adjacency matrix of G.

$$A_G(u, v) = \begin{cases} l(e), & \text{if } e_{uv} \in E \\ 0, & \text{otherwise} \end{cases} \tag{2.1}$$

$\bigoplus$ denotes an aggregated concatenation operation on a set of elements. For example, $\bigoplus_{x \in X} x$ denotes a long vector, matrix, or tensor, which is the result of concatenating all the elements in X. $[\mathbf{a} \quad \mathbf{b}]$ denotes the concatenate **a** and **b**. $vec(\cdot)$ is the vectorization (flatten) function for matrix,

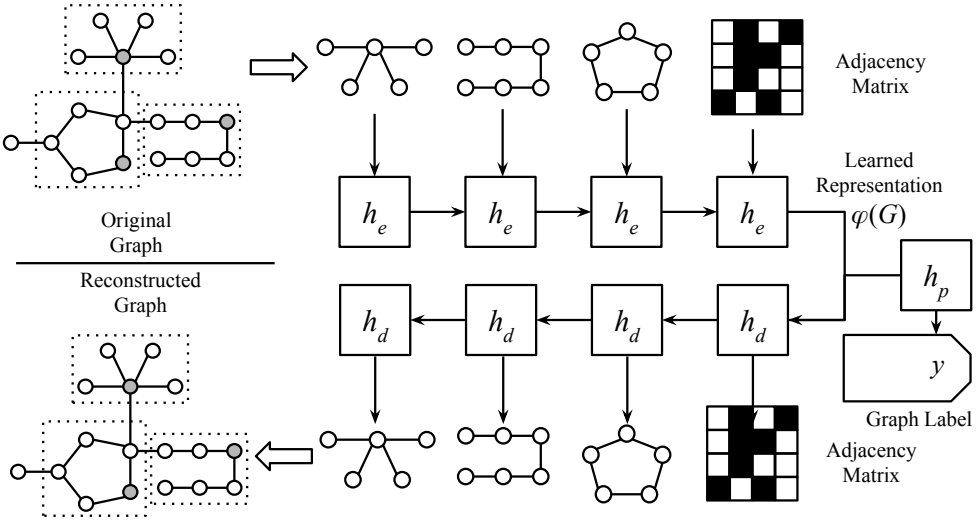


Figure 2.1: Discriminative Graph Autoencoder

it concatenate each column of the matrix to become a column vector. $\circ$ denotes the function composition operation.

Based on the notations described above, we formally define our problem as follows: *Given a set of graphs $\mathbb{G}$ and their labels Y, the goal is to map each graph $G_i \in \mathbb{G}$ to a d-dimensional vector representation $\boldsymbol{h}_i \in \mathbb{R}^d$, i.e., Learning a mapping function $\varphi : \{G, y\} \rightarrow \boldsymbol{h}$ which produces informative representations of each graph G while preserving the discriminative information according to y.*

2.3 Discriminative Graph Autoencoder

In this section, the proposed method Discriminative Graph Autoencoder (DGA) is introduced to deal with the previously mentioned challenges. While some recent work [14, 34] has put their focus on addressing the sparsity problem, they can only provide a pairwise similarity matrix for all graph pairs instead of a vector representation of each graph. Applications are thus constrained to kernel-based algorithms like SVM instead of general data mining algorithms, e.g. neural networks, nor any visualization can be done, which requires low-dimensional vector representations.

Our approach tackles the sparsity problem and produces vector representations simultaneously. The key ideas of DGA are shown in Figure 2.1. Given an input graph G to be encoded, we

first decompose it into several subgraphs. These subgraphs and the adjacency matrix of G are then inputted into an encoder network. The output of the encoder corresponds to the vector representation of G we intend to learn, i.e., $\varphi(G)$. To preserve the graph structures and the discriminative information of labels, $\varphi(G)$ is used for predicting the graph label y of G and is further input into a decoder network to reconstruct the original input subgraphs and adjacency matrix. Equipped with this model architecture, the mapping function $\varphi(\cdot)$ could be decomposed into two functions as follows:

$$\varphi = h_e \circ F. \tag{2.2}$$

where $F(\cdot)$ denotes the mapping function for graph sampling and $h_e(\cdot)$ represents the encoding function.

Graph sampling is an effective way to collect information from a graph. The graph kernels only seek predefined substructures and use their frequencies of occurrence to represent a graph. Graphs that only contain few of the predefined substructures may obtain sparse representations, making the performance of similarity measurement less effective. In addition, some graphs may contain substructures beyond those predefined substructures since the list is not comprehensive, thus the corresponding information cannot be captured by the representations. Our key idea of the graph sampling function $F(G)$ is to use arbitrary subgraphs found in each graph to represent the graphs instead of searching for certain predefined types of substructures, which preserves more structural information comparing to frequency-based approaches. For each input graph, a certain number of subgraphs are extracted around the selected vertices in the graph, which are combined with the adjacency matrix of the entire graph to prepare the input to autoencoder.

The autoencoding is used to capture the structural information from the sampled inputs for representation learning. It can learn a smooth representation and avoid the sparsity problem. To capture discriminative information in the graph autoencoder, the key idea of the proposed algorithm is to let the autoencoder reconstruct the input and predict the graph label y_i from the learned representation $\varphi(G_i)$ at the same time. It is achieved by defining a new objective function instead of using the original autoencoder objective function which purely minimizing the reconstruction

error between the input and output.

2.3.1 Graph Sampling

Our goal of graph sampling is to sample the raw structural data from the graph G and pack the information in a matrix $F(G)$ with a predefined shape. A natural way of graph sampling is to use adjacency matrices which may contain structural information of the graphs. However, two problems prevent us from using this solution. First, adjacency matrices are usually of different sizes. Take data flow graphs of computer programs as an example. Different programs may contain different numbers of variables corresponding to different numbers of vertices, resulting in different sizes of adjacency matrices. Second, adjacency matrices are vulnerable to changes in the vertex order. Two graphs of the same structure may have significant differences in their adjacency matrices if their vertices are in different orders.

To tackle the challenges, we propose a graph sampling method that uses subgraphs together with the adjacency matrix to present the information in a graph as shown in Equation 2.3.

$$F(G) = \Big[\bigoplus_{v \in C} (f \circ g)(v) \quad vec(A_G) \Big]^T, \tag{2.3}$$

where $\bigoplus$ is the concatenation operation on all the vectors in a set, $f(\cdot)$ is the vectorization operation on a single subgraph, and $g(\cdot)$ is the subgraph extraction operation around the given vertex $v \in C$, C is a specially selected subset of vertex set V, $vec(\cdot)$ is the matrix vectorization function, A_G is the adjacency matrix of graph G.

$F(G)$ is a matrix consists of multiple vectors. Each of the vectors encodes the information of a vectorized subgraph except the last vector, which encodes the information of the truncated adjacency matrix of G. A_G is part of $F(G)$ because the autoencoder needs to have an overview of the entire graph and the relationship between the subgraphs. It is also essential for reconstructing the entire graph after decoding.

$F(\cdot)$ is decomposed into the selection of central vertices C, subgraph extraction $g(\cdot)$, and vectorization function $f(\cdot)$, which are introduced in the following sections.

2.3.1.1 Vertex Selection

A set of vertices C is selected as the central vertices for subgraph extraction. Before the selection, it is necessary to put an order to the information in the graph which is essential for the following encoding part. Even two similar graphs if their vertices are arranged in a different order, the adjacency matrices could be not similar at all. The goal of sorting the vertices is to decrease the difference in the adjacency matrices of the graphs and subgraphs when the graphs are similar to each other, and increase the difference in the adjacency matrices of the graphs and subgraphs when they are not similar. The goal is achieved by canonical labeling [47]. After the canonical labeling, it is easier to tell whether graphs are similar or not from their adjacency matrices. Canonical labeling gives each vertex v in the graph $G = \{V, E\}$ a unique index $p(v) \in \{x|1 \leq x \leq |V|\}$, $p(v_i) = p(v_j)$ if and only if $i = j$. The indices are the ranks of vertices sorted in canonical order. It is an indication of relations between vertices in two isomorphic graphs. For example, G and $G' = \{V', E'\}$ are isomorphic, $v_i \in V$ should be identical to $v'_j \in V'$ if $p(v_i) = p'(v'_j)$.

BLISS [48] is one of the most famous algorithms in graph isomorphism. We follow their idea to index the vertices in the graph according to their structural information of the graphs without the label information $l(v)$ or $l(e)$, which makes our method more general. Some graphs already have clear orders of the vertices within the graphs, for which the indexing process is not necessary.

With the label $p(v)$ we are able to select the central vertices set C according to Equation 2.4.

$$C = \{v|p(v) = ax + 1, 0 \leq x \leq s - 1, x \in \mathbb{Z}\}, \tag{2.4}$$

$$\begin{aligned} a = \underset{i \in \mathbb{Z}}{\operatorname{argmax}} (s-1) * i + 1, \\ \text{s.t.} (s-1) * i + 1 \leq |V|, \end{aligned} \tag{2.5}$$

where a is the gap between two selected vertices in the sequence of vertices sorted by their $p(v)$, s is the user-defined size of central vertices set C. The vertices are selected with a common interval of a. The goal is to evenly distribute the selected vertices in the graph. In Equation 2.5, $(s-1) * i + 1$ is the index of the last vertex in C. a is the interval between the indices of the selected vertices so

that the vertices are evenly separated. It is selected to maximize the index of the last vertex in C to ensure the indices of the selected vertices are distributed in V.

2.3.1.2 Subgraph Extraction

Subgraph extraction function $g(\cdot)$ is defined to extract a subgraph around a given vertex.

Algorithm 1 Subgraph Extraction

```
1:  Input: G, central, r
2:  Output: R
3:  queue ← PriorityQueue
4:  queue.comparator ← Vertex Comparator
5:  queue ← {central}
6:  visited(central) ← TRUE
7:  while |R| < r do
8:      u ← queue.pop()
9:      R.add(u)
10:     for v in u's neighbour do
11:         if not visited v then
12:             queue.push(v)
13:             visited(v) ← TRUE
14:         end if
15:     end for
16: end while
17: Return subgraph(R)
```

In Algorithm 1, it shows how one subgraph is extracted from G with a selected vertex as $central$ and a user-defined size $|V_{g(v)}| = r$, where $V_{g(v)}$ is the vertex set of the extracted subgraph. It is similar to a breadth-first search (BFS) with a priority queue optimization. The search starts from the vertex $central$ and ends when the number of vertices reaches the subgraph size limit r. The priority queue is always able to select the next vertex by comparing their $p(v)$ value, which can always break the tie between vertices. From line 2 to 5, the queue is initialized with a single object $central$ in it. From line 6 to 15, the subgraph keeps expanding until reaches r vertices. From line 7 to 8, the priority queue with vertex comparator pops out the best vertex candidate u and adds it to

the subgraph according to the following rules. First, it selects the vertices with the shortest distance from the $central$. Second, it selects the v which equals to $\text{argmin}_{v \in V} p(v)$ to further break the ties. From line 9 to 14, it pushes the neighbors of u into the priority queue as potential candidates for subgraph expanding.

2.3.1.3 Vectorization

As shown in Equation 2.6, the vectorization function $f(\cdot)$ takes a subgraph as input and put its information into a compatible form with the autoencoder h_e.

$$f(g) = \left[vec(A_g)^T \quad (\bigoplus_{v \in V_g} l(v))^T \right]^T, \tag{2.6}$$

where g is the input graph, A_g is the adjacency matrix of g, $\bigoplus$ is the concatenate operation on a set of elements, V_g is the vertex set of g. Notably, the order of the vertices in g is first sorted according to $p(v)$ where $v \in V_g$. The adjacency matrix A_g and the concatenate operation is all sorted according to the vertex order. The final output of $f(g)$ is a matrix, the first part of which is the vectorized adjacency matrix A_g. The rest of the row is the sequence of the labels of the vertices in the subgraph. Therefore, the size of the final output is $|V_g| \times |V_g| + |V_g|$. It contains all the information we need for the subgraph and ready to be encoded.

2.3.2 Autoencoding

A straightforward architecture for the autoencoder is multi-layer perceptron [49], However, it is not flexible with different sizes of the input and cannot leverage discriminative information. We use it as one of our baseline methods in the experiments. Thus, we proposed a novel architecture with a separate branch to leverage the discriminative information of the graph labels. Motivated by the "sequence to sequence learning model" [50], two LSTM [51] networks are used to better fit the various sizes of the graphs.

As shown in Figure 2.1, the discriminative autoencoder consists of the encoder $h_e(\cdot)$, decoder $h_d(\cdot)$, and predictor $h_p(\cdot)$. After graph sampling, the vectorized subgraphs of graph G is input into

the encoder h_e. The encoder produces a low dimensional smooth vector representation of G denoted as $\varphi(G)$. $\varphi(G)$ is input to the predictor h_p to produce a label prediction for G denoted as $\hat{y}$. It is also input to h_d to reconstruct the original input graph. The two branches starting from the learned representation corresponds to two terms, $\mathcal{L}_1$ and $\mathcal{L}_2$, in the objective function,. $\mathcal{L}_1$ is defined for minimizing the autoencoder reconstruct error for the information in the input graph G. $\mathcal{L}_2$ is defined for minimizing the error of predicting the label of the input graph G from the learned representation $\varphi(G)$.

2.3.2.1 Encoding

For the encoder LSTM network, the vectorized information $F(G)$ of a graph of shape ($s+1$, $r*(r+1)$) is input to LSTM in $s+1$ steps, where $s=|C|$ is the number of subgraphs extracted, $r=|V_g(v)|$ is the size of each graph. Then, among the $s+1$ output of the encoder LSTM network, only the last one is collected. This collected output is the learned representation $\varphi(G)$.

2.3.2.2 Decoding

$\varphi(G)$ is used by the decoder LSTM to reconstruct the input. The decoder LSTM network uses $\varphi(G)$ as the input of the first step. For the rest s steps of the LSTM decoder, only padding vectors filled with ones are used as input. So there are total $s+1$ steps for the input to the decoder LSTM. To reconstruct the $s+1$ subgraphs input to the encoder, all $s+1$ outputs are collected from the decoder.

The non-discriminative graph autoencoder is optimized by minimizing the following objective function:

$$\mathcal{L}_1(\theta) = -\frac{1}{n}\sum_{i=1}^{n} L((h_d \circ h_e \circ F)(G), F(G)) \tag{2.7}$$

where $L(\cdot,\cdot)$ is the categorical cross-entropy. θ is the parameter set of the autoencoder. The output subgraphs should be in reverse order to ease the optimization of the autoencoder [12].

With the decoded adjacency matrix and the information of the subgraphs. The graph can be reconstructed in its original space. Apply the graph sampling procedure on the adjacency matrix

to map the vertices to the vertices in the subgraphs, so that the reconstructed vertex labels in the subgraphs can be mapped to the graph.

2.3.2.3 Discriminative Information

Besides the structural information of the input graph G, the label information y could also be used to increase the rate of discriminative information in the learned representations among graphs.

To leverage discriminative information, the learned representation $\varphi(G)$ is used as shown in Equation 2.8.

$$\mathcal{L}_2(\theta) = -\frac{1}{n}\sum_{i=1}^{n} L((h_p \circ h_e \circ F)(G_i), \mathbf{y}_i), \tag{2.8}$$

where $\mathbf{y}$ is the binarized label of the graphs. $\mathbf{y}_{i,j} = 1$ if $y_i = j$, otherwise $\mathbf{y}_{i,j} = 0$, θ is the parameter set of the discriminative autoencoder, $h_p(\cdot)$ is a single-layer perceptron predictor use softmax as activation function, $L(\cdot, \cdot)$ is categorical cross-entropy. Optimizing against this loss function would force the autoencoder to focus on the information that can distinguish one class of graphs from the other classes, instead of treating each class of graphs equally.

2.3.2.4 DGA Objective Function

The autoencoder is optimized against the sum of two loss functions for representation learning, which is balanced by the parameter λ shown in Equation 2.9. The overall objective function requires the learned representations not only contain as much information in the extracted subgraphs, but also incorporate the discriminative information in graph labels.

$$\begin{aligned}\mathcal{L}(\theta) = -\frac{1}{n}\sum_{i=1}^{n} & L((h_d \circ h_e \circ F)(G_i), F(G_i)) \\ & + \lambda L((h_p \circ h_e \circ F)(G_i), \mathbf{y}_i).\end{aligned} \tag{2.9}$$

2.3.3 Time Complexity

The time complexity of Discriminative Graph Autoencoder is analyzed as follows. The complexity of the discriminative autoencoder depends on many complex parameters. Moreover, it is not the

Table 2.1: Datasets Statistics

Datasets \ Statistics	$\lvert G\rvert$	$\overline{\lvert V\rvert}$	$\overline{\lvert E\rvert}$	Class
MUTAG	188	17	39	2
NCI1	4110	29	64	2
PTC	344	25	51	2
PROTEIN	1113	39	145	2

bottleneck of the proposed method. Therefore, the time complexity analysis focuses on the graph sampling part. The complexity of the vertex indexing is $O(|V|)$, which is the complexity of BLISS algorithm. The complexity for a simple breadth-first search to extract a subgraph of size r is $O(r)$. However, a priority queue is used for finding the best vertices which put additional complexity to the method. The time complexity of each push or pop operation of the priority queue is $O(\log r)$. There are r operations in total, where r is the size of the subgraph. Therefore, the complexity for extracting one subgraph is $O(r \log r)$. Vectorize one subgraph would cost $O(r^2)$, which is the size of the output vector $f(g)$. The total complexity of $f(g(v))$ is $O(r^2 + r \log r) = O(r^2)$. So the complexity for generating s subgraphs is $O(r^2 s)$. Thus, the overall complexity of the graph sampling is $O(|V| + r^2 s)$.

2.4 Experiments

We empirically evaluate the representations learned from the proposed model DGA on two different tasks, i.e., classification and visualization. Three questions are mainly analyzed: (1) How efficient is DGA in learning graph representations? (2) How effective are the learned representations for graph classification? (3) How informative is the visualization of the learned vector representations in low dimensional space?

2.4.1 Experimental Setup

2.4.1.1 Datasets

The datasets used are benchmark datasets for graph classification, which consist of MUTAG, NCI1, PROTEIN, and PTC. MUTAG [52], NCI1[53], and PTC [54] are datasets of chemical compounds. PROTEIN [55] is a dataset of the protein structures. The number of examples for each class is balanced. The statistics information of these datasets are shown in Table 2.1, where $|G|$ is the number of graphs in each dataset, $\overline{|V|}$ and $\overline{|E|}$ denote the average number of vertices and edges. $Class$ is the number of different classes in the labels of the graphs in the datasets.

2.4.1.2 Baselines

Four different types of baseline methods are used for comparison as follows. First, Graphlet Kernels (GK) [24], Shortest-Path graph kernels (SP) [26], fast subtree kernels (WL) [17] are used as traditional baselines. They are frequency-based methods of counting subgraphs, vertex pairs, and subtrees respectively. Second, the advanced baselines are Deep Graph Kernels [34] and PCSN [43]. Deep Graph Kernels are three graph kernels derived from the three traditional approaches above, namely DGK, DSP, and DWL. Third, we also implemented a naive approach of Multi-Layer Perceptron autoencoder (MLP), which only uses the adjacency matrix as input. Fourth, to show the effectiveness of discriminative information, a non-discriminative graph autoencoder (GA) is implemented, which only uses $\mathcal{L}_1$ as the loss function.

2.4.1.3 Parameter Setting

For three traditional baseline methods, we follow the parameter setting in the original paper. For deep graph kernels, the length of the subgraph embedding is set to 128. For PCSN, we set $w = k = 12$, which are the sizes of the sampling information in the graphs. The following parameters are set to DGA for the experiments. 1. Number of subgraphs selected $s = 12$. 2. Size of each subgraph $r = 12$. 3. The length of the final embedding is 128. 4. $\lambda = 1$ in the loss function $\mathcal{L}(\theta)$. These parameters are set based on cross-validation, the parameter analysis.

Table 2.2: Time For Graph Sampling

Datasets \ Methods	PCSN	DGA
MUTAG	00.83s	00.75s
NCI1	25.15s	21.36s
PTC	01.74s	01.61s
PROTEIN	10.60s	08.70s

2.4.2 Efficiency

The efficiency of graph sampling is evaluated and compared with the state-of-the-art method PCSN [43] on four datasets shown in Table 2.2. We follow similar experimental settings in [43]. Since the complexity of LSTM autoencoder is much lower than the graph sampling process, the evaluation is mainly targeting the efficiency of graph sampling, which is potentially the bottleneck of the efficiency of the entire process. The number of subgraphs and the size of the subgraphs are set to 12 for both of the methods.

The total time for PCSN and DGA in graph sampling and network input generating is shown in Table 2.2. The results show that Discriminative Graph Autoencoder performs slightly better than PCSN on all benchmark datasets. Our proposed DGA method is very efficient since it is very careful in using the expensive traditional graph isomorphism algorithm as subroutines. It uses vertex indexing only once and without considering the labels. The subgraph sampling is also boosted with priority queue optimization.

2.4.3 Graph Classification

Graph classification aims at assigning unlabeled graphs into target categories based on available labeled training graphs. In this section, we evaluate DGA on the four datasets stated above with the graph classification task.

For each of the algorithms to be tested, the following steps are conducted to produce the results. First, the datasets are divided to conduct a 10-fold cross-validation. Second, during each fold, each

Table 2.3: Classification Accuracy with Standard Deviation

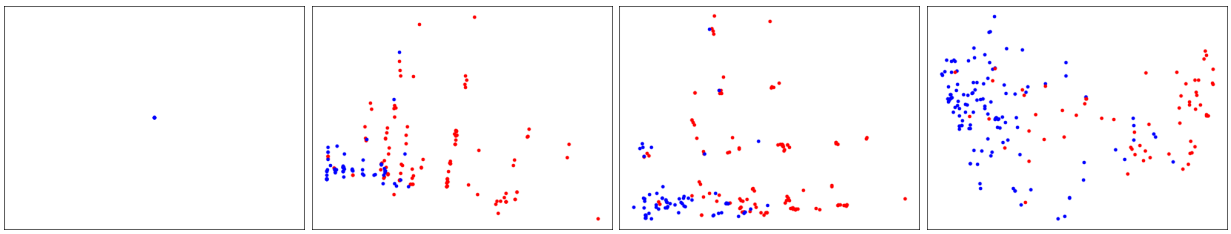
	MUTAG	NCI1	PTC	PROTEIN
GK	81.66±2.11	62.28±0.29	57.26±1.41	71.67±0.55
SP	85.22±2.43	73.00±0.24	58.24±2.44	75.07±0.54
WL	83.48±6.51	80.13±0.50	56.97±2.01	72.92±0.56
DGK	82.66±2.11	62.48±0.25	57.32±1.13	71.68±0.50
DSP	87.44±2.72	73.55±0.51	60.08±2.55	75.68±0.54
DWL	82.94±2.68	**80.31±0.46**	59.17±1.56	73.30±0.82
PCSN	92.63± 4.21	78.59± 1.89	60.00± 4.82	75.89± 2.76
MLP	76.05±9.71	67.05±2.39	59.31±4.96	73.76±3.54
GA	92.57±5.84	72.74±1.75	70.93±3.89	77.45±3.29
DGA	**93.63±5.21**	74.55±1.46	**71.24±4.60**	**77.71±2.37**

of the algorithms is trained on the corresponding training dataset and then used to convert all the testing graphs into vector representations. Finally, an SVM is trained on the converted training dataset (i.e., labeled vector representations) and tested on the converted testing dataset.

Table 2.3 shows the accuracy of all the experiments, which is defined as the quotient of the number of correctly classified instances divided by the total number of instances in the testing dataset. From the results, we can see that our method has the highest accuracy on most of the datasets. DGA has a significant increase in accuracy in PTC. By comparing the GA and DGA, we can see the power of discriminative information in the representations. DGA has an extra loss function of $\mathcal{L}_2$ to keep the representation focusing more on the discriminative information in the graphs. By this loss function, the accuracy rises on four of the datasets.

2.4.4 Graph Visualization

One useful application of graph representation learning is to produce meaningful visualizations that layout graphs in a low-dimensional space. The visualization performance is also an indicator of the quality of the representations. DGA is compared with three traditional baselines, GK, SP, and WL, on MUTAG dataset, and achieved a significant increase in this task. Deep graph kernels and PCSN are not selected in this experiment as they are not capable of generating vector



(a) GK (b) SP (c) WL (d) DGA

Figure 2.2: Visualization of Learned Representations

representations of graphs. To achieve the best performance for the baselines during the experiment, they are first trained to produce high dimensional representations. Then, PCA is used for reducing the dimensionality of these representations to 2. The proposed method DGA is directly trained to produce a 2-D representation. The representations learned from all four methods are visualized in a 2-D space shown in Figure 2.2.

Each point in the figure is the projection of one representation learned from a graph in the dataset. The colors of the points correspond to the class labels $L(G)$ of the graphs. In Figure 2.2a, the GK cannot provide valid visualization of the learned representations since all instances are mapped to one point. This is mainly because the learned representations are long and extremely sparse, which results in a concentration effect after PCA. SP and WL methods have better performances in visualization which are shown in Figure 2.2b and Figure 2.2c. However, it clearly shows striped or grid patterns in the distributions of points. The main reason is the vector space of the learned representations is not smooth. The representations are rigid integer coordinations representing headcounts for substructures and cannot represent the fine-grained information in graphs. The representations are so rigid and stick to the integer coordinated grids in the hyperspace that even after the PCA dimensionality reduction the grids can still be seen. The visualization results using the representations generated from DGA is shown in Figure 2.2d. Comparing to other methods, it has three advantages: (1) it can directly produce valid low-dimensional data representation; (2) the vector space of the learned representations is smooth, which avoids the damage caused by the rigidity in the representations; (3) it can use the discriminative information to better visualize the

differences between different classes of graphs.

3. EFFICIENT NEURAL ARCHITECTURE SEARCH WITH NETWORK MORPHISM AND BAYESIAN OPTIMIZATION*

The training time of a neural network is significantly longer than most of the shallow models. It makes neural architecture search takes much longer time than traditional hyperparameter tuning for shallow models since it searches through a large number of neural network instances and trains each of them completely. In this chapter, we propose a novel method to make use of the weights in previously trained neural networks to boost the training of a new neural network during the neural architecture search process.

3.1 Introduction

Existing NAS algorithms are usually computationally expensive. The time complexity of NAS is $O(n\bar{t})$, 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 [6, 56, 57, 58, 59], gradient-based methods [60, 61, 62] and evolutionary algorithms [63, 64, 65, 66, 67, 68], require a large n to reach a good performance. Moreover, many of them train each of the n neural networks from scratch, which is very slow.

Initial efforts have been devoted to making use of network morphism in neural architecture search [69, 70]. It is a technique to morph the architecture of a neural network but keep its functionality [71, 72]. 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 towards 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 an operation from the network morphism

*Reprinted with permission from "Auto-Keras: An Efficient Neural Architecture Search System" by Haifeng Jin, Qingquan Song, Xia Hu, 2019. Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 1946–1956, Copyright 2019 by Association for Computing Machinery.

operation set to morph an existing architecture to a new one. The network morphism-based NAS methods are not efficient enough. They either require a large number of training examples [69], or inefficient in exploring the large search space [70]. How to perform an efficient neural architecture search with network morphism remains a challenging problem.

As we know, Bayesian optimization [73] 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 [8, 73, 74, 75, 76, 77], 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 the 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 non-trivial to design a Bayesian optimization method for network morphism-based NAS due to the following challenges. First, the underlying Gaussian process (GP) is traditionally used for learning the probability distribution of functions in Euclidean space. To update the Bayesian optimization model with observations, the underlying GP is to be trained with the searched architectures and their performances. 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, in the context of network morphism, it is not to maximize a function in Euclidean space, but finding a node in a tree-structured search space, where each node represents a neural architecture and each edge is a morph operation. Thus traditional gradient-based methods cannot be simply applied. Third, the changes caused by a network morphism operation is complicated. 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 a challenging problem.

In this chapter, 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 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 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 network morphism is defined to address the changes in the neural architectures based on layer-level network morphism. The proposed approach is compared with the state-of-the-art NAS methods [70, 78] 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 have developed a widely adopted open-source AutoML system based on our proposed method, namely AutoKeras. It is an open-source AutoML system, 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 chapter are as follows:

- Propose an algorithm for efficient neural architecture search based on network morphism guided by Bayesian optimization.
- Conduct intensive experiments on benchmark datasets to demonstrate the superior performance of the proposed method over the baseline methods.
- Develop an open-source system, namely AutoKeras, which is one of the most widely used AutoML systems.

3.2 Problem Statement

The general neural architecture search problem we studied in this chapter is defined as: Given a neural architecture search space $\mathcal{F}$, the input data D divided into D_{train} and D_{val}, and the cost function $Cost(\cdot)$, we aim at finding an optimal neural network $f^* \in \mathcal{F}$, which could achieve the lowest cost on dataset D. The definition is equivalent to finding f^* satisfying:

$$f^* = \underset{f \in \mathcal{F}}{\operatorname{argmin}} \, Cost(f(\boldsymbol{\theta}^*), D_{val}), \tag{3.1}$$

$$\boldsymbol{\theta}^* = \underset{\boldsymbol{\theta}}{\operatorname{argmin}} \, \mathcal{L}(f(\boldsymbol{\theta}), D_{train}). \tag{3.2}$$

where $Cost(\cdot, \cdot)$ is the evaluation metric function, *e.g.*, accuracy, mean sqaured error, $\boldsymbol{\theta}^*$ is the learned parameter of f.

The search space $\mathcal{F}$ covers all the neural architectures, which can be morphed from the initial architectures. The details of the morph operations are introduced in 3.3.3. Notably, the operations can change the number of filters in a convolutional layer, which makes $\mathcal{F}$ larger than methods with fixed layer width [61].

3.3 Network Morphism Guided by Bayesian Optimization

The key idea of the proposed method is to explore the search space via morphing the neural architectures guided by Bayesian optimization (BO) 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 performances; (2)Generation: generate the next architecture to observe by optimizing a delicately defined acquisition function; (3) Observation: obtain the actual performance by training the generated neural architecture. There are three main challenges in designing a method for morphing the neural architectures with Bayesian optimization. We introduce three key components separately in the subsequent sections coping with the three challenges.

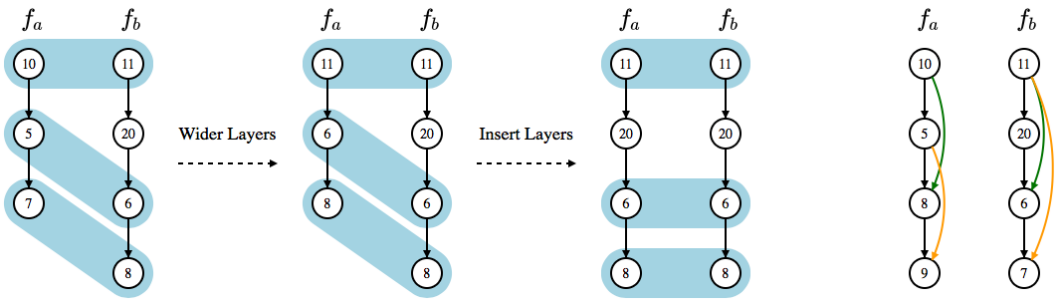


Figure 3.1: Neural Network Kernel. Given two neural networks f_a, f_b, and matchings between the similar layers, the figure shows how the layers of f_a can be changed to the same as f_b. Similarly, the skip-connections in f_a also need to be changed to the same as f_b according to a given matching.

3.3.1 Edit-Distance 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 traditional Gaussian process (GP). 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. More edits needed from one architecture to another means the further distance between them, thus less similar they are. The proof of the validity of the kernel function is presented in Appendix B.

3.3.1.1 Kernel Definition

Suppose f_a and f_b are two neural networks. Inspired by Deep Graph Kernels [34], we propose an edit-distance kernel for neural networks. Edit-distance here means how many operations are needed to morph one neural network to another. The concrete kernel function is defined as:

$$\kappa(f_a, f_b) = e^{-\rho^2(d(f_a, f_b))}, \tag{3.3}$$

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 [79], which ensures 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 [80]. Based on the search space $\mathcal{F}$ defined in Section 3.2, we tackle the problem by proposing an approximated solution as follows:

$$d(f_a, f_b) = D_l(L_a, L_b) + \lambda D_s(S_a, S_b), \tag{3.4}$$

where D_l denotes the edit-distance for morphing the layers, *i.e.*, the minimum edits 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 between the distance of the layers and the skip-connections.

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:

$$D_l(L_a, L_b) = \min \sum_{i=1}^{|L_a|} d_l(l_a^{(i)}, \varphi_l(l_a^{(i)})) + \Big||L_b| - |L_a|\Big|, \tag{3.5}$$

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 (3.6),

$$d_l(l_a, l_b) = \frac{|w(l_a) - w(l_b)|}{max[w(l_a), w(l_b)]}, \tag{3.6}$$

where $w(l)$ is the width of layer l.

The intuition of Equation (3.5) is consistent with the idea of network morphism shown in Figure 3.1. Suppose a matching is provided between the nodes in two neural networks. The sizes

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 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 edit-distance of the layers is defined as D_l in Equation (3.5).

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:

$$\boldsymbol{A}_{i,j} = max[\boldsymbol{A}_{i-1,j} + 1, \boldsymbol{A}_{i,j-1} + 1, \boldsymbol{A}_{i-1,j-1} + d_l(l_a^{(i)}, l_b^{(j)})], \tag{3.7}$$

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 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 3.1, the skip-connections with the same color are matched pairs. Similar to $D_l(\cdot,\cdot)$, $D_s(\cdot,\cdot)$ is defined as follows:

$$D_s(S_a, S_b) = \min \sum_{i=1}^{|S_a|} d_s(s_a^{(i)}, \varphi_s(s_a^{(i)})) + \Big||S_b| - |S_a|\Big|, \tag{3.8}$$

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:

$$d_s(s_a, s_b) = \frac{|u(s_a) - u(s_b)| + |\delta(s_a) - \delta(s_b)|}{max[u(s_a), u(s_b)] + max[\delta(s_a), \delta(s_b)]}, \tag{3.9}$$

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 (3.8) 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 weighted bipartite graph matching problem is solved by the Hungarian algorithm (Kuhn-Munkres algorithm) [81].

3.3.1.2 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 (3.4) 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 (3.3) 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 B. □

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 [82]. Any finite metric space distance can be isometrically embedded into Euclidean space by changing the scale of the distance measurement [83]. By using Bourgain theorem [79], 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. □

3.3.2 Optimization for Tree Structured Space

The second challenge of using Bayesian optimization to guide network morphism is the optimization of the acquisition function. The traditional acquisition functions are defined in Euclidean

space. The optimization methods are not applicable to the tree-structured search via network morphism. To optimize our acquisition function, we need a method to efficiently optimize the acquisition function in the tree-structured space. To deal with this problem, we propose a novel method to optimize the acquisition function on tree-structured space.

Upper-confidence bound (UCB) [84] is selected as our acquisition function, which is defined as:

$$\alpha(f) = \mu(y_f) - \beta\sigma(y_f), \tag{3.10}$$

where $y_f = Cost(f, D)$, β is the balancing factor, $\mu(y_f)$ and $\sigma(y_f)$ are the posterior mean and standard deviation of variable y_f predicted by the Gaussian process. It has two important properties, which fit our problem. First, it has an explicit balance factor β for exploration and exploitation. Second, $\alpha(f)$ is directly comparable with the cost function value $c^{(i)}$ in search history $\mathcal{H} = \{(f^{(i)}, \boldsymbol{\theta}^{(i)}, c^{(i)})\}$. The UCB estimates the lowest possible cost given the neural network f. $\hat{f} = argmin_f \alpha(f)$ is the generated neural architecture for next observation.

The tree-structured space is defined as follows. During the optimization of $\alpha(f)$, $\hat{f}$ 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. Using network morphism for NAS may end up with a very large architecture without enough exploration on the smaller architectures. However, in our tree-structure search, we 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 comparatively small architectures.

Inspired by 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 [85] and simulated annealing [86] is proposed. A* algorithm is widely used for tree-structure search. It maintains a priority queue of nodes and keeps expanding the best node in the queue. Since A* always exploits the best node, simulated annealing is introduced to balance the exploration and exploitation by not selecting the estimated best architecture with a probability.

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

As shown in Algorithm 2, the algorithm takes minimum temperature T_{low}, temperature decreasing rate r for simulated annealing, and search history $\mathcal{H}$ described in Section 3.2 as the 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 6, the searched architectures are pushed into 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 18, it 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 popped 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 of whether it is pushed into the queue 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 from line 14 to 16, which record the minimum acquisition function value and the corresponding architecture.

3.3.3 Graph-Level Network Morphism

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.

Follow the four network morphism operations on a neural network $f \in \mathcal{F}$ defined in [70], 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 is adding an additive connection from node u to node v denoted as $add(G, u, v)$. The fourth 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. 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.

3.3.4 Time Complexity Analysis

As described at the start of Section 3, Bayesian optimization can be roughly divided into three steps: update, generation, and observation. The bottleneck of the algorithm efficiency 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.

3.4 AutoKeras

Based on the proposed neural architecture search method, we developed an open-source AutoML system, namely AutoKeras. It is named after Keras [87], which is known for its simplicity in creating neural networks. Similar to SMAC [8], TPOT [88], Auto-WEKA [74], and Auto-Sklearn [76], the goal is to enable domain experts who are not familiar with machine learning technologies to use machine learning techniques easily. However, AutoKeras 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, cloud services are not free to use, which may not be affordable for everyone who wants to use AutoML techniques. Second, the cloud-based AutoML usually requires complicated configurations of Docker containers and Kubernetes, which is not easy for people without a rich computer science background. Third, the AutoML service providers are honest-but-curious [89], 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 bridge the gap, we developed AutoKeras.

It is challenging, to design an easy-to-use and locally deployable system. First, we need a concise and configurable application programming interface (API). For the users who don't have rich experience in programming, they could easily learn how to use the API. For the advanced users, they can still configure the details of the system to meet their requirements. Second, local computation resources may be limited. We need to make full use of the local computation resources to speed up the search. Third, the available GPU memory may be of different sizes in different environments. We need to adapt the neural architecture sizes to the GPU memory during the search.

3.4.1 System Overview

The system architecture of AutoKeras is shown in Figure 3.2. We design this architecture to fully make use of the computational resource of both CPU and GPU, and utilize the memory efficiently by

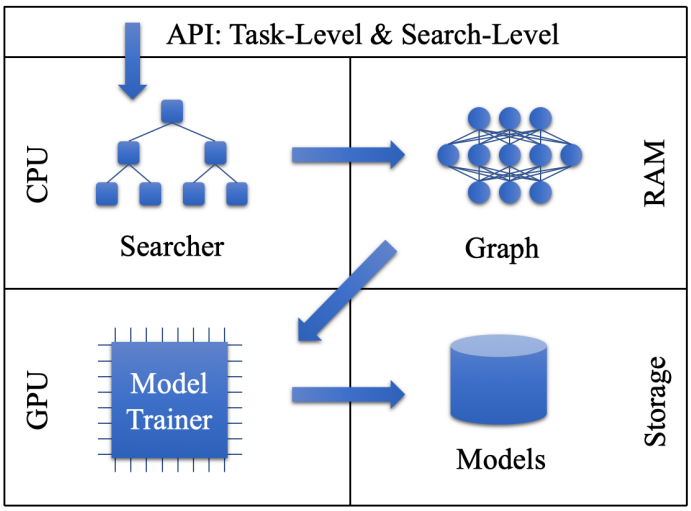


Figure 3.2: AutoKeras System Overview. (1) The user calls the API. (2) The Searcher generates neural architectures on CPU. (3) Graph builds real neural networks with parameters on RAM from the neural architectures. (4) The neural network is copied to GPU for training. (5) Trained neural networks are saved on storage devices. The Searcher is updated based on the training results. Step (2) to (5) will repeat until it reaches the time limit.

only placing the currently useful information on the RAM, and save the rest on the storage devices, *e.g.*, hard drives. 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 Searcher is the module of the neural architecture search algorithm containing Bayesian Optimizer and Gaussian Process. These search algorithms run on CPU. The Model Trainer is a module responsible for the computation on GPUs. It trains given neural networks with the training data in a separate process for parallelism. The Graph is the module processing the computational graphs of neural networks, which is controlled by the Searcher for the network morphism operations. The current neural architecture in the Graph is placed on RAM for faster access. 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 the storage devices.

A typical workflow for the AutoKeras system is 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. The Bayesian Optimizer in the Searcher would generate a new

architecture using CPU. It calls the Graph module to build the generated neural architecture into a real neural network in the RAM. The new neural architecture is copied the GPU for 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.

3.4.2 Application Programming Interface

The design of the API follows the classic design of the Scikit-Learn API [90, 91], which is concise and configurable. The training of a neural network requires as few as three lines of code calling the constructor, the fit and predict function respectively. To accommodate the needs of different users, we designed two levels of APIs. The first level is named as task-level. The users only need to know their task, *e.g.*, Image Classification, Text Regression, to use the API. The second level is named search-level, which is for advanced users. The user can search for a specific type of neural network architectures, *e.g.*, multi-layer perceptron, convolutional neural network. To use this API, they need to preprocess the dataset by themselves and know which type of neural network, *e.g.*, CNN or MLP, is the best for their task.

Several accommodations have been implemented to enhance the user experience with the AutoKeras 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 AutoKeras 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.

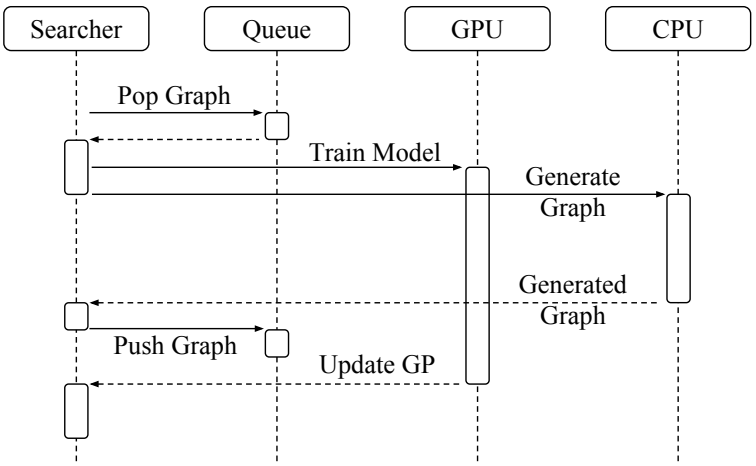


Figure 3.3: CPU and GPU Parallelism. The Searcher obtains the next neural architecture to be trained and starts the training on GPU in a separate process. Then, instead of waiting for the training to finish, it directly starts to generate the next neural architecture on CPU.

3.4.3 CPU and GPU Parallelism

To make full use of the limited local computation resources, 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 sequential order. The GPUs will be idle during the update and generation. The CPUs will be idle during the observation. To improve 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 3.3 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 requests 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.

3.4.4 GPU Memory Adaption

The size of the neural networks needs to be limited according to the GPU memory. Otherwise, the system would crash because of running out of GPU memory. Many approaches have been taken to search for memory-efficient neural architectures [92]. In AutoKeras, we implement 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 generated neural networks.

3.5 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 is gained from Bayesian optimization and network morphism? 3) Does the proposed kernel function correctly measure the similarity among neural networks in terms of their actual performance?

For more details on the experimental setup and implementation please refer to Appendix A.

Three benchmark datasets, MNIST [11], CIFAR10 [93], and FASHION [94] are used in the experiments to evaluate our method. They prefer very different neural architectures to achieve good performance.

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 convolutional layers and the width of those layers.

- Conventional Methods: SPMT [73] and SMAC [8]. 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 [70], NASBOT [78]. 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.

In addition, the performance of the deployed system of AutoKeras (AK-DP) is also evaluated in the experiments. The difference from the AK above is that AK-DP uses various advanced techniques to improve the performance including learning rate scheduling, multiple manually defined initial architectures.

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. Notably, AK-DP uses a real deployed system setting, whose result is not directly comparable with the rest of the methods. Except for AK-DP, all other methods are fairly compared using the same initial architecture to start the search.

Table 3.1: 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%**
AK-DP	0.60%	3.60%	6.72%

3.5.1 Evaluation of Effectiveness

We first evaluate the effectiveness of the proposed method. The results are shown in Table 3.1. The following conclusions can be drawn based on the results.

(1) AK-DP is evaluated to show the final performance of our system, which shows the deployed system (AK-DP) achieved state-of-the-art performance on all three datasets.

(2) The proposed method AK achieves the lowest error rate on all the three datasets, which demonstrates that AK is able to find simple but effective architectures on small datasets (MNIST) and can explore more complicated structures on larger datasets (CIFAR10).

(3) 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.

(4) 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 in training the neural architectures 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.

(5) 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 a much longer time, which limits the number of architectures it can search within a given time.

3.5.2 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 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. The two figures in Figure 3.4 shows the same results but with different X-axes. The Y-axis is the lowest error rate achieved. The X-axes are the number of neural networks searched 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 the left part of Figure 3.4, it shows BFS

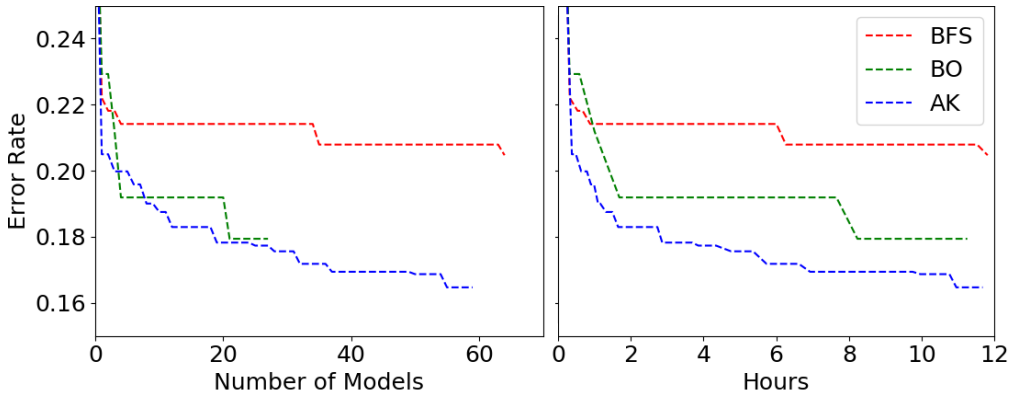
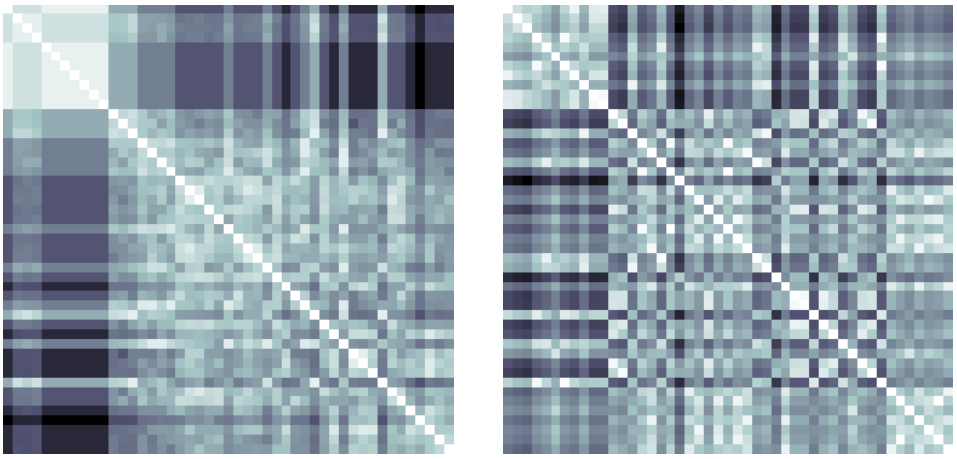


Figure 3.4: Evaluation of Efficiency. The two figures plot the same result with different X-axis. BFS uses network morphism. BO uses Bayesian optimization. AK uses both.

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 neural architectures than BFS because of higher time complexity.

Two conclusions can be drawn by comparing BO and AK. First, network morphism does not negatively impact search performance. In the left part of Figure 3.4, when BO and AK search a similar number of neural architectures, they achieve similar lowest error rates. Second, network morphism increases training efficiency, thus improve the performance. As shown in the left part of Figure 3.4, 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 the right part of Figure 3.4. At the end of the searching time, AK achieves lower error rate than BO.

To show the quality of the edit-distance neural network kernel, we investigate the difference between the two matrices $\boldsymbol{K}$ and $\boldsymbol{P}$. $\boldsymbol{K} \in \mathbb{R}^{n \times n}$ is the kernel matrix, where $\boldsymbol{K}_{i,j} = \kappa(f^{(i)}, f^{(j)})$. $\boldsymbol{P} \in \mathbb{R}^{n \times n}$ describes the similarity of the actual performance between neural networks, where $\boldsymbol{P}_{i,j} = -|c^{(i)} - c^{(j)}|$, where $c^{(i)}$ is the cost function value in the search history $\mathcal{H}$ described in Section 3.3. 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 $[-1, 1]$.



(a) Kernel Matrix (b) Performance Similarity

Figure 3.5: Kernel and Performance Matrix Visualization. (a) shows the proposed kernel matrix. (b) is a matrix of similarity in the performance of the neural architectures.

The difference between $\boldsymbol{K}$ and $\boldsymbol{P}$ are measured quantitatively with mean square error, which is 1.12×10^{-1}.

$\boldsymbol{K}$ and $\boldsymbol{P}$ are visualized in Figure 3.5a and 3.5b. Lighter color means larger values. There are two patterns can be observed in the figures. First, the white diagonal of Figure 3.5a and 3.5b. 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 3.5a. 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 3.5b, which indicates that when the kernel measures two architectures as similar, they tend to have similar performance.

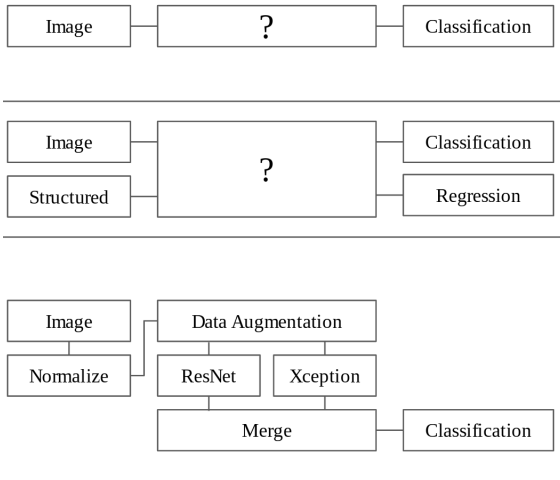
Neural architecture search is the most important component of automated deep learning but not the only component. For a deep learning solution, besides the neural architecture, there are many other hyperparameters to tune, for example, the type of optimizer or the learning rate. Moreover, these hyperparameters are correlated with the neural architecture instead of independent from each other. How to tune these hyperparameters together with the neural architecture is the key problem if neural architecture search is to be used in an overall automated deep learning process. In this chapter, we propose a new AutoML framework to map the neural architecture and other hyperparameters into the same hyperparameter space. In addition, we also propose a search algorithm with warm-start in the hyperparameter tuning process to improve the efficiency of the framework.

4.1 Introduction

From an application perspective, people adopting deep learning would like to receive a complete deep learning solution as the output, which includes not only the tuned neural network but also the preprocessing steps and optimizers with tuned hyperparameters. Moreover, practitioners prefer more efficient AutoML systems. In other words, they would like to find a good and complete deep learning solution with fewer trials.

To address such requirements, we developed a new version of AutoKeras, an AutoML library for deep learning. It automates the process of model selection, hyperparameter tuning, and neural architecture search. It encapsulates the end-to-end process from raw datasets to trained machine learning models into an extremely simple and flexible interface. Novice users can implement deep learning models with a few lines of code, while the advanced users can also easily customize different parts of the model to their needs. It implements a greedy algorithm to warm-start the search space and fine-tunes the best model in the warm-start model list, which can find a good solution for a given task with less number of trials.

AutoKeras specializes in raw data types like images and texts in addition to structured data,




```
1 import autokeras as ak
2 # Task API
3 auto_model = ak.ImageClassifier()
4 auto_model.fit(x_train, y_train)
5 result = auto_model.predict(x_test)
```

```
6 # Input/Output API
7 auto_model = ak.AutoModel(
8   inputs=[ak.ImageInput(), ak.StructuredDataInput()],
9   outputs=[ak.ClassificationHead(), ak.RegressionHead()])
10 auto_model.fit(x=[x_image, x_struct], y=[y_clf, y_reg])
```

```
11 # Functional API
12 input_node = ak.ImageInput()
13 output_node = ak.Normalization()(input_node)
14 output_node = ak.ImageAugmentation()(output_node)
15 output_node_1 = ak.ResNetBlock(version='v2')(output_node)
16 output_node_2 = ak.XceptionBlock()(output_node)
17 output_node = ak.Merge()([output_node_1, output_node_2])
18 output_node = ak.ClassificationHead()(output_node)
19 auto_model = ak.AutoModel(input_node, output_node)
20 auto_model.fit(x_train, y_train)
```

Figure 4.1: Three Levels of APIs

which is supported by existing AutoML libraries [74, 76, 88, 95]. It is also flexible enough to cover multi-modal data and multi-task use cases. AutoKeras is built base on Keras Tuner [96], Keras [87], and TensorFlow [97]. The models created by AutoKeras can be easily exported as Keras models, which can be deployed in various production environments with the help of the TensorFlow ecosystem.

4.2 API Design

The API design of AutoKeras follows the style of Keras, which is well received by the deep learning community. It has three levels of APIs, namely, task API, IO API, and functional API, ranging from the simplest to the most configurable. The code of using these APIs are shown in Figure 4.1 with diagrams showing the corresponding neural network models. The parts with question marks are tuned automatically.

The task API requires the least amount of configurations from the user. As shown in Figure 4.1 from line 3 to 5, an example of the image classification task is implemented within three lines of code. Six different tasks are supported in task APIs, including classification and regression for image, text, and structured data.

The IO API (input/output API) supports multi-modal data and multi-task use cases. In Figure 4.1

from line 7 to 10, the dataset is a set of images with attributes, for example, an image of a house with attributes describing the total area and location of the house. Each data sample is associated with two prediction targets, a label for classification and a real value for regression. The user needs to specify the inputs and outputs format of the model as shown in line 8 and 9. The training data are passed in lists in the same order in line 10.

The functional API enables advanced users to tailor the search spaces according to their needs. It resembles the Keras functional API to let the user build the computational graph of the deep learning model with the building blocks. The example from line 12 to line 19 connects both preprocessing steps and neural network blocks, which applies data normalization and data augmentation to the data before passing it to a neural network with ResNet [98] and XceptionNet [99]. Notably, on line 15, the version of the ResNet is specified as v2, which further reduces the size of the search space. There are many such configurable hyperparameters for other blocks as well. They are tuned automatically if left unspecified. Moreover, the users can also create custom neural network blocks to use with the functional API.

AutoKeras is fully compatible with the TensorFlow and Keras ecosystem. The fit function in AutoKeras supports all the arguments supported by the Keras fit function. The model found by AutoKeras can be easily exported as a Keras model. With the help of the TensorFlow ecosystem, it is ready for deployment in various production environments.

4.3 System Architecture

In this section, we introduce the system architecture of AutoKeras, which is explained from two perspectives: the core workflow of running an AutoML task in the system and important components of the system.

4.3.1 Core Workflow

Figure 4.2 shows the core workflow of the AutoKeras system, which is drawn in data flow diagram format. The core AutoKeras workflow consists of the following steps. First, AutoKeras analyzes the training data to determine e.g. whether a column in structured data is categorical or

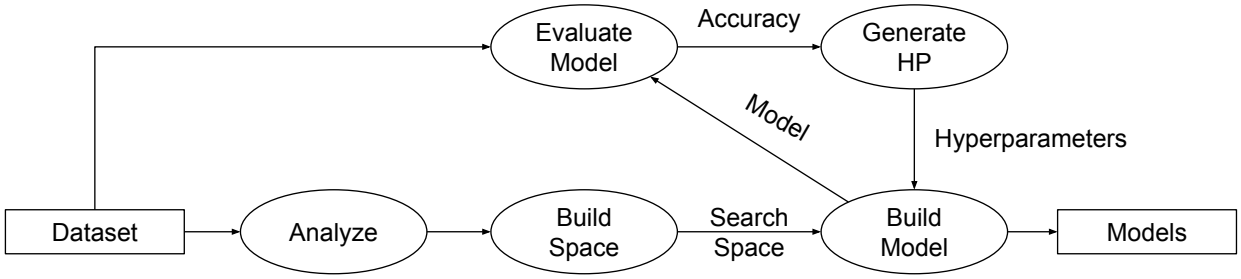


Figure 4.2: The Core Workflow of AutoKeras System

numerical, whether the image data contains the channel dimension, or whether the classification labels need to be encoded. Second, it uses this information to construct a suitable search space that encompasses both neural architecture patterns and common hyperparameters. Third, it goes through the search loop. Every time the search algorithm would generate a set of hyperparameter values to build a model from the search space. The model is trained on the training set and evaluated on the validation set. The result is sent back to the search algorithm. Finally, all the searched models are stored on disk.

4.3.2 Components

AutoKeras uses Keras and TensorFlow to build machine learning models. Keras Tuner, a hyperparameter tuning framework for Keras, provides the infrastructure for implementing the search space and the searching algorithm in AutoKeras.

Figure 4.3 shows the major components of the AutoKeras system. To make the figure more illustrative, the dependencies of AutoKeras (TensorFlow, Keras, Keras Tuner) are also shown in the figure. Each box represents a class. Arrows with solid lines represent class extensions. For example, the Block class extends the HyperModel class. Arrows with dashed lines represent usages. For example, the Block class uses the Layer class in Keras.

As we introduced in the API design section, user can directly access the classes in the AutoModel box, which provides the APIs to the users. The API classes use the AutoTuner class for searching the model and Graph class as the overall search space. The AutoTuner extends the Tuner class in Keras Tuner, which is responsible for managing the search process like instantiating the model,

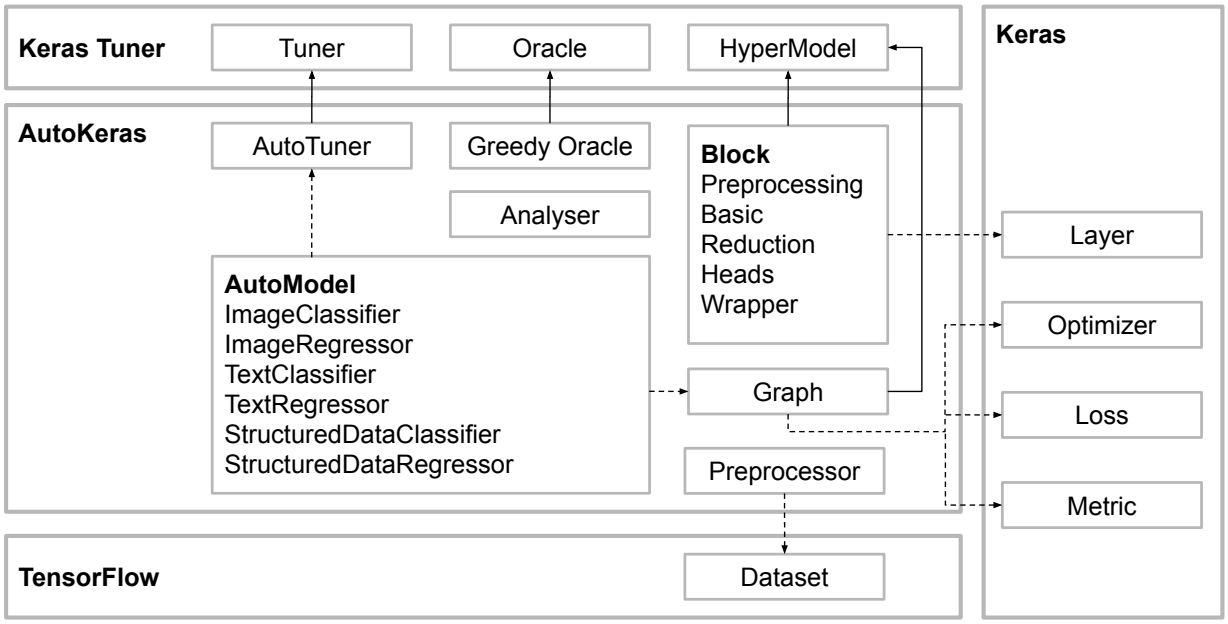


Figure 4.3: Important Components of AutoKeras System

fitting the model on training data, and tracking the evaluation results. Inside the tuner, the Greedy Oracle, which extends in the Oracle class in Keras Tuner, is responsible for receiving the evaluation results and generating new hyperparameter values.

The Graph class extends the HyperModel class in Keras Tuner, which is responsible for defining a search space. When instantiated to a Keras model, it uses the optimizer, loss, metrics from Keras. A search space can be built into a model with a set of hyperparameter values. The overall search space in Graph consists of smaller building blocks, which are implemented in with subclasses of the Block class, which all extends the HyperModel class since the building blocks are also search spaces. The building blocks can be sorted into 5 different categories. The preprocessing blocks are for preprocessing steps like data normalization, data augmentation, or categorical feature encoding. The basic blocks are the commonly used neural network blocks like convolutional blocks or fully-connected blocks. The reduction blocks are to reshape the output tensors of the previous blocks into vectors for compatibility to the following blocks. The head blocks are the output heads for specific tasks like classification and regression. The wrapper blocks are model selection blocks for specific tasks, for example, a block for image data is a model selection between ResNet, XceptionNet, and

vanilla convolutional networks. All these blocks when instantiated to models will use the Layer classes in Keras.

There are also some other components in AutoKeras. The Analyser class is to analyze the data before the search starts. The Preprocessor class is to preprocess the data before feeding it into the model, for example, encoding the classification labels. It uses the TensorFlow Dataset class for manipulation of the data, which is capable of large scale streaming datasets.

4.4 Methodology

In this section, we introduce how we map all the hyperparameters into the same search space and the details of the search algorithm.

4.4.1 Search Space

The search space of AutoKeras includes the state-of-the-art neural network models for the supported tasks. For models like EfficientNet [100] and BERT [101], pretrained weights can be applied. Besides the neural networks, it also tunes the hyperparameters from the preprocessing steps and the training process, for example, image data augmentation, text vectorization, categorical feature encoding, optimizer, learning rate, and weight decay.

We map all these hyperparameters into the same space including the neural architectures. Despite the complexity in the neural architecture search space, it can be represented by conditional hyperparameters. For example, we can use one hyperparameter to decide how many skip connections in a neural network. For each skip-connection, we use two additional hyperparameters to decide its starting point and end point. The value of one hyperparameter decides the number of other hyperparameters. We designed the hyperparameter space supporting such hyperparameter usages. Therefore, all hyperparameters, no matter it belongs to the neural architecture or not, can be treated equally by the search algorithm. In this way, we enabled hyperparameter joint tuning with the neural architectures.

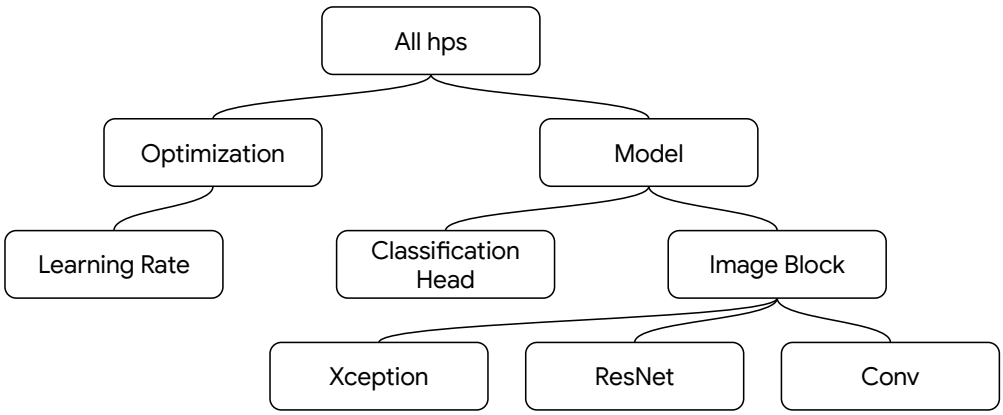


Figure 4.4: The Hierarchy of Hyperparameters

4.4.2 Search Algorithm

AutoKeras implements a greedy search algorithm, which starts from a list of predefined models that are known to have good performance and exploit them. It is greedy since it always selects the current best model and generates new models in its neighborhood.

The process of the algorithm can be summarized as follows. First, it iterates through a list of models to evaluate the target dataset. Second, it selects the current best model and builds a hyperparameter tree from it. The leaves in the tree are hyperparameters. An example of the hierarchy of hyperparameters is shown in Figure 4.4. The subtrees represent different parts of the model like ResNet and classification head, the leaves of which are not shown in the figure but exist. The learning rate is an actual hyperparameter, so it is a leaf. Third, it generates a new hyperparameter value set by replacing the values of subtree leaves. The subtree is selected according to a probability distribution, which considers both the dependency relation between the hyperparameters and the size of the subtree. The less number of leaves a subtree contains, the more likely it is selected. Therefore, the search algorithm prefers exploitation on the neighborhood of a good model to exploration over new models. The new values are randomly generated. Fourth, go back to the second step and repeats until reach the maximum number of trials set by the user.

Dataset	MNIST	CIFAR10	IMDB	Titanic	Housing
Task	ImageClf	ImageClf	TextClf	StructDataClf	StructDataReg
Metric	Accuracy	Accuracy	Accuracy	Accuracy	MSE
State of the Art	99.82%	98.90%	96.80%	≈100%	0.23
AutoKeras	99.04%	97.10%	93.93%	82.20%	0.28
AutoGluon	98.70%	96.79%	85.70%	82.58%	0.28
AutoKeras Days	0.51	1.8	1.2	0.002	0.06
AutoGluon Days	0.08	6.0	0.05	0.0001	0.0002

Table 4.1: Experimental Results

4.5 Experiments

Our experiments evaluate AutoKeras performance on some of the most widely-used benchmark datasets: MNIST [11], CIFAR10 [93], IMDB Reviews [102], Titanic [103], and California Housing [104]. We measure accuracy for the classification tasks and mean squared error for the regression tasks. As a baseline, we compare results to AutoGluon [105], an established AutoML solution developed by Amazon. The training data is used for both model training and the model search process, while the test data is kept exclusively for evaluating the final model to avoid "test set overfitting". To obtain these results, no other configuration options were passed to AutoKeras in addition to the training data; meanwhile, AutoGluon required manually specifying the number of training epochs. We used a single Nvidia Tesla V100 GPU with 16GB of memory to search over 10 models for each experiment.

Experimental results are shown in Table 4.1. For image and text classification tasks, AutoKeras outperformed the baseline method and achieved results close to state-of-the-art solutions [100, 106, 107]. For structured data tasks, AutoKeras achieved similar results to the baseline method, which are not as good as the state-of-the-art solutions [108, 109]. Comparing with image or text classification, more expert-provided domain knowledge is involved when designing the state-of-the-art solution for structured data tasks. It highlights a limitation of AutoML: the inability to leverage information other than the provided dataset. Lastly, for CIFAR10, the running time of AutoKeras is significantly lower than the AutoGluon baseline due to leveraging an adaptive number of epochs. For the

IMDB Reviews, Titanic, and California Housing datasets, the difference in running time is due to differences in the search spaces.

5. AUTOMATED DATA AUGMENTATION

Among all the modules of automated deep learning, one module stands out for its large number of hyperparameters to tune and its importance in the overall process, which is data augmentation. The data augmentation step contains a large number of operations to select from, which increases the size of the search space exponentially. In this chapter, we propose a novel method for automated data augmentation, which does not need to go through the search loop to select the hyperparameters, but augment the data with a dynamic data augmentation strategy in a one-shot style.

5.1 Introduction

Data augmentation is a technique to create synthetic data from existing data with controlled perturbation. For example, in the context of image recognition, data augmentation refers to applying image operations, e.g., cropping and flipping, to input images to generate augmented images, which have labels the same as their originals. In practice, data augmentation has been widely used to improve the generalization in deep learning models and is thought to encourage model insensitivity towards data perturbation [98, 110, 111]. Although data augmentation works well in practice, designing data augmentation strategies requires human expertise, and the strategy customized for one dataset often works poorly for another dataset. Recent efforts have been dedicated to automating the design of augmentation strategies. It has been shown that training models with a learned data augmentation policy may significantly improve test accuracy [1, 112, 113, 114, 115].

However, we do not yet have a good theory to explain how data augmentation improves model generalization. Currently, the most well-known hypothesis is that data augmentation improves generalization by imposing a regularization effect: it regularizes models to give consistent outputs within the vicinity of the original data, where the vicinity of the original data is defined as the space that contains all augmented data after applying operations that do not drastically alter image features [116, 117, 118]. Previous automated data augmentation works claim that the performance gain from applying learned augmentation policies arises from the increase in diversity [113, 114,

119]. However, the "diversity" in the claims remains a hand-waving concept: it is evaluated by the number of distinct sub-policies utilized during training or visually evaluated from a human perspective. Without formally defining diversity and its relation to regularization, the augmentation strategies can only be evaluated indirectly by evaluating the models trained on the augmented data, which may cost thousands of GPU hours [119]. It motivates us to explore the possibility of using an explicit diversity measure to quantify the regularization effect of the augmented data may have on the model. Thus, we can directly maximize the diversity of the augmented data to strengthen the regularization effect to improve the generalization of the model.

There are many existing work aiming at solving the auto data augmentation problem. The first work that tried to learn data augmentation policy from data is AutoAugment (AA) [119]. Specifically, AutoAugment utilizes a recurrent neural network (RNN) as the controller to find the best policy in a separate search process on a small proxy task (smaller model size and dataset size). Once the search process is over, the learned policies are transferred to the target task and fixed during the whole training process. These learned augmentation policies significantly improve the generalization of deep models [119]. However, its search time is huge: it costs roughly 5,000 GPU hours to search for the best policies on a smaller dataset they call "reduced CIFAR-10", which consists of 4,000 randomly chosen images.

Most of the following works adopted the AutoAugment search space and formulation with improved optimization algorithms [1, 112, 114, 115]. Population-based augmentation (PBA) [114] replaces the fixed policy with a dynamic schedule of policies evolving along with the training process. Fast AutoAugment (Fast AA) [112] proposes a "density match" method to accelerate the search process and treats the augmented data as missing points in the training set. RandAugment (RA) [113] eliminates the separate search process by randomly applying augmentation sub-policies, which best resembles our work. Adversarial AutoAugment (Adv. AA) [1] achieves state-of-the-art results by utilizing an RNN controller to learn policies that could generate augmented data with higher loss.

In this chapter, we mathematically show that training models on augmented data impose a

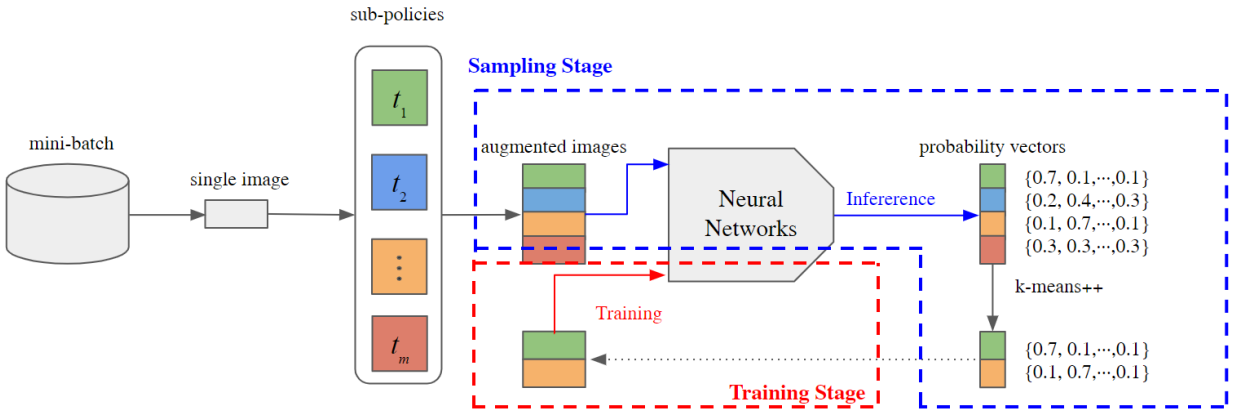


Figure 5.1: The DivAug framework overview. At the sampling stage, each data in the mini-batch is augmented by multiple randomly generated sub-polices. Notice the probability vectors of these augmented data are also obtained. Then k-means++ seeding algorithm is used to sample a subset of augmented data whose probability vectors are far apart from each other and thus diversifies the augmented data. At the training stage, the generated data is used to train the model.

regularization effect. Namely, the loss implicitly contains a data-driven regularization term that is in proportion to the variance of probability vectors, where probability vectors are the outputs from models trained with the augmented data. From above, we measure the diversity of a set of augmented data by the variance of their corresponded probability vectors. Based on the measure, we propose a search-free automated data augmentation framework named **DivAug**. As illustrated in Figure 5.1, the framework has two stages: the sampling stage, where we automate the data augmentation process according to the diversity measure, and the training stage, where we train the model using the augmented data. Specifically, at the sampling stage, for each image, we sample a subset of augmented images with high diversity by applying the k-means++ seeding algorithm [120], where the augmented data accompanied with probability vector which is far away from that of the original data is sampled with high probability.

Following our mathematical derivation, the regularization effect increases with the diversity of the augmented data. Consequently, the stronger regularization effect can lead to better model generalization, which is observed in terms of improved model performance. Our main contributions can be summarized as follows:

- We mathematically show that, if a model is trained on augmented data, a regularization term,

which is proportional to the variance of probability vectors, can be decomposed from the loss function

- From this, we propose a new measure for quantifying the diversity of augmented data. We validate in our experiments that the relative gain in the accuracy of a model after applying data augmentation is highly correlated to our proposed measure.
- Based on the proposed measure, we design an sampling-based framework to explicitly maximize diversity. Without requiring a separate search process, the performance gain from DivAug is comparable to the state-of-the-art method with better efficiency.
- Our method is search-free and unsupervised. We show that our method can further boost the performance of the semi-supervised learning algorithm, making it highly applicable to real-world problems, where labeled data is scarce.

Table 5.1: Summary of automated data augmentation.

Method	non-fixed	search-free	unsupervised	without proxy tasks
AA [119]	✗	✗	✗	✗
Fast AA [112]	✗	✗	✗	✓
PBA [114]	✓	✗	✗	✗
Adv. AA [1]	✓	✗	✗	✓
RA [113]	✗	✓	✓	✓
DivAug (proposed method)	✓	✓	✓	✓

As shown in Table 5.1, we outline a general taxonomy of automated data augmentation methods, characterized by four core properties. *Non-fixed*: augmentation policies are dynamically changed along with the training process; *search free*: automated data augmentation methods do not require a separate search process; *unsupervised*: automated data augmentation methods do not require label information to find the best policy; and *without proxy tasks*: automated data augmentation methods perform the search directly on target tasks.

5.2 Methodology

In this section, we introduce the design and implementation of DivAug. First, we describe our search space in Section 5.2.1. Then we mathematically show that after employing augmented data, the training loss implicitly contains a data-driven regularization term that is in proportion to the variance of probability vectors (Section 5.2.2). Subsequently, we propose to measure the diversity of a set of augmented data by the variance of their corresponded probability vectors. Based on the measure, we derive a sampling-based automated data augmentation method to explicitly maximize the diversity of augmented data (Section 5.2.3).

5.2.1 Search Space

We adopt the basic structure of the well-designed search space introduced in AutoAugment [119]. There are totally 16 image operations in our search space, including `Sharpness`, `ShearX/Y`, `TranslateX/Y`, `Rotate`, `AutoContrast`, `Invert`, `Equalize`, `Solarize`, `Posterize`, `Color`, `Brightness`, `Cutout` [121], `Sample Pairing` [122], and `Contrast`. Let $\mathcal{O} = \{\texttt{Sharpness}, \cdots, \texttt{Contrast}\}$ be the set of all available operations. Each operation $\text{op} \in \mathcal{O}$ has two parameters: p, the probability of applying the operation; and m, the magnitude of the operation. To avoid creating confusion in notations, we use $\overline{\text{op}}(\cdot; m)$ to represent image transformation specified by op, with magnitude m. Given an image x, the operation $\text{op}(x; p, m)$ is defined as:

$$\text{op}(x; p, m) = \begin{cases} \overline{\text{op}}(x; m), & \text{with probability } p. \\ x, & \text{with probability } 1 - p. \end{cases}$$

Each operation comes with a maximum range of magnitudes to avoid extreme image transformations. For example, `Rotate` operation is only allowed to rotate images at most 30 degrees. The maximum range of magnitude for each operation is set to be the same as those reported in the AutoAugment. Meanwhile, we normalize the magnitude parameter m to within $[0, 1]$, where 1 stands for the maximum acceptable magnitude. One example for illustrating the operation is shown in Figure 5.2.

In general, previous automated data augmentation methods search for the top augmentation

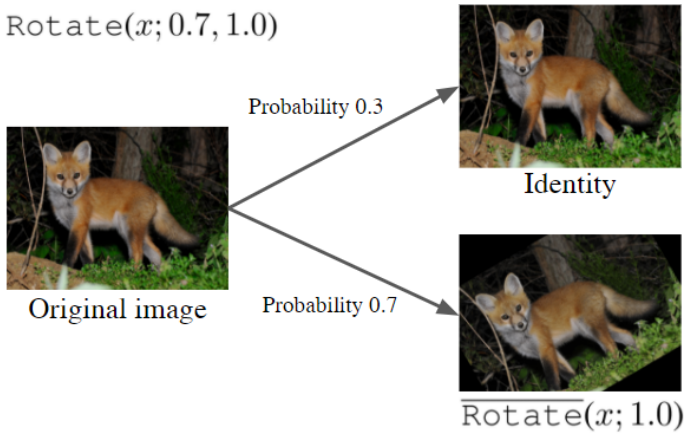


Figure 5.2: The schema of operation Rotate(·;0.7,1.0), where 1.0 is the normalized magnitude of the operation. Notice Rotate(·;0.7,1.0) denotes rotating the image by 30 degrees with the probability of 0.7.

policy, which is a set of five sub-policies, with each sub-policy consisting of two operations to be applied to the original images in sequence. Let t be the sub-policy that consists of two consecutive operations, namely, $t(x) = \mathrm{op}_2(\mathrm{op}_1(x; p_1, m_1); p_2, m_2)$. For the sake of description convenience, we simplify the notation as $t := \mathrm{op}_2 \circ \mathrm{op}_1$. Given the search space, previous automated data augmentation methods explore and rank the possible policy candidates in a separate search process. Once the search process is over, the top five policies are collected to form a single final policy, which is a set containing 25 distinct sub-policies. The final policy is fixed throughout the training process. For each image in a mini-batch, only one sub-policy will be randomly selected to be applied [119].

However, the fixed policy may be sub-optimal due to the following two factors. First, there does not exist a sub-policy universally better than all other sub-policies throughout the training process [1, 114, 123]. For example, sub-policies that can reduce generalization error at the end of training is not necessarily a good sub-policy at the initial phase [124]. Second, the choices (hence diversity) of the augmented data is limited by the fixed set of unique sub-policies. From the above analysis, we design our search space similar to the previous methods with two differences. First, to introduce more stochasticity, we relax both the probability p and magnitude m as continuous parameters with value range $[0, 1]$. Second, the final policy in our search space is defined as the

universal set that contains all the possible sub-policies. In contrast, the final policy in other work's search space is set to a fixed set of 25 unique sub-policies. We note that RandAugment [113] samples the sub-policies uniformly over the search space similar to ours. The major distinctions in RandAugment are 1) the magnitude parameter m is fixed discrete integer value, 2) the probability parameter p is fixed to 1. That means RandAugment *always* applies operations on the original data.

5.2.2 Regularization Effects of Data Augmentation

We start by introducing the setting and notations of representation learning. Consider a neural network $f_\theta(x)$ parameterized by θ (*italic* for vectors and **bold** for matrices). f_θ map the input x into a vector representation $f_\theta(x) \in \mathbb{R}^D$ with D output dimensions. We aim to minimize loss functions $l : \mathbb{R}^D \times \mathbb{R} \to \mathbb{R}$ over a dataset $\{(x_i, y_i)\}_{i=1}^N$, where $y_i \in \{1, \cdots D\}$. Let $\hat{p}(y|x) = \texttt{Softmax}(f_\theta(x))$ be the probability vector, where the `Softmax` function is used to normalize the neural network's output $f_\theta(x)$ into a probability distribution. We denote the loss function to be minimized as $L = \sum_{i=1}^N L_i$, where $L_i = l(\hat{p}(y|x_i), y_i)$. We denote the gradient of l with respect to the first argument as $l' \in \mathbb{R}^D$. Similarly, we use $l'' \in \mathbb{R}^{D \times D}$ to represent the Hessian matrix of l with respect to the first argument. We use t to represent the sub-policy, and T is the set of all available sub-policies. x_i^t is the augmented data in the vicinity of x_i obtained by applying t to x_i. We use $\langle \cdot, \cdot \rangle$ to denote inner-product. For a set $\mathcal{S}$, we use $|\mathcal{S}|$ to represent its cardinality. With these notations, after applying data augmentation, the new loss function becomes:

$$L_i' = \mathbb{E}_{t \sim \mathrm{T}}[l(\hat{p}(y|x_i^t), y_i)]. \tag{5.1}$$

Suppose data augmentation does not significantly modify the feature map. Using the first order Taylor approximation, we can expand Equation (5.1) around point ψ_i:

$$\begin{aligned} L_i' \approx & l(\psi_i, y_i) + \\ & \mathbb{E}_{t \sim \mathrm{T}}[\langle \hat{p}(y|x_i^t) - \psi_i, l'(\psi_i, y_i) \rangle]. \end{aligned} \tag{5.2}$$

The second term in Equation (5.2) can be cancelled by picking $\psi_i = \mathbb{E}_{t\sim \mathrm{T}}\hat{p}(y|x_i^t)$, i.e., ψ_i is the averaged probability vector of all samples within the vicinity of x_i. If we further expand Equation (5.1) around point $\psi_i = \mathbb{E}_{t\sim \mathrm{T}}\hat{p}(y|x_i^t)$ by considering the second order term, we have:

$$L_i' \approx l(\psi_i, y_i) + \mathbb{E}_{t\sim \mathrm{T}}[\Delta_i^\top l''(\psi_i, y_i)\Delta_i]. \tag{5.3}$$

$\Delta_i := \hat{p}(y|x_i^t) - \psi_i$ is the difference between the probability vector $\hat{p}(y|x_i^t)$ referring to the augmented data x_i^t, and the averaged probability vector ψ_i. The second term in Equation (5.3) is so called the "data-driven regularization term", which is exact the variance of the probability vector $\hat{p}(y|x_i^t)$, weighted by $l''(\psi_i, y_i)$. That means employing augmented data imposes a regularization effect by implicitly controlling the variance of model's outputs.

5.2.3 The DivAug Framework

To establish the relationship between the diversity of augmented data and their regularization effect, we propose a new diversity measure, called Variance Diversity, for the augmented data whose regularization effect can be quantified. Based on this, we derive a sampling-based framework that explicitly maximizes the Variance Diversity of the augmented data.

5.2.3.1 Diversity Measure of Augmented Data

We start by proposing a new diversity measure for augmented data, whose regularization effect can be quantified.

From Equation (5.3), after training models on augmented data, a data-driven regularization term can be decomposed from the loss function [*]. From above, we quantify the diversity of a set of augmented data by the variance of their corresponding probability vectors. Formally, given a model f_θ, for a set of augmented data $\mathcal{S} = \{x^{t_j}\}_{j=1}^S$, where x^{t_j} is generated from the same original data x

[*]This regularization effect is also found in [117] and has been validated empirically. The main difference between our setup and those in [117] is that the model assumes to be a general kernel classifier in [117].

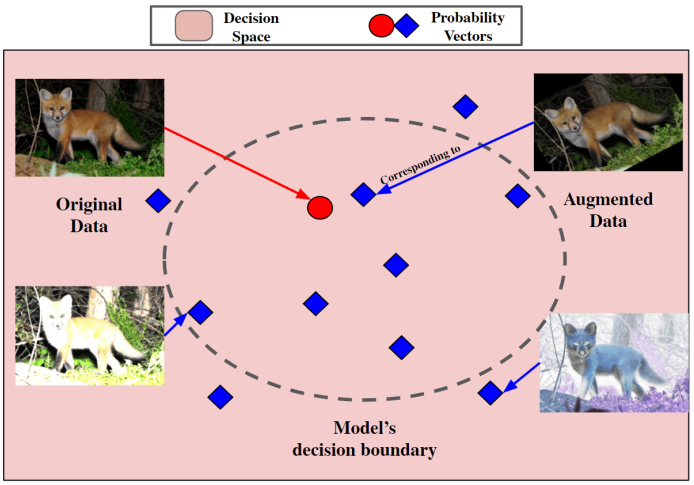


Figure 5.3: An example to illustrate the diversity between augmented data. DivAug explicitly looks for augmented data whose corresponding probability vectors are far away from each other in the decision space.

by applying different sub-policy t_j, we define the diversity of $\mathcal{S}$ as:

$$\mathcal{D}(\mathcal{S}) = \mathbb{E}_{x^{t_j} \in \mathcal{S}} \Delta^\top \Delta. \tag{5.4}$$

$\hat{p}(y|x^{t_j}) := \texttt{Softmax}(f_\theta(x^{t_j}))$ is the probability vector corresponding to x^{t_j}, and $\Delta = \hat{p}(y|x^{t_j}) - \mathbb{E}_{x^{t_j} \in \mathcal{S}} \hat{p}(y|x^{t_j})$. Actually, if $l''(\psi_i, y_i)$ in Equation (5.3) is set as the identity matrix, the diversity of augmented data is exact the data-driven regularization term in Equation (5.3). According to Equation (5.4), we name our diversity measure "Variance Diversity". We note that this is a unsupervised model-specific measure, which depends only on the model prediction without involving any label information.

Intuitively, as illustrated in Figure 5.3, if a set of augmented data has large Variance Diversity, that means their corresponding probability vectors are far away from each other. Therefore, it is harder for models to give consistent predictions for diversely augmented data. This forces the models to generalize over the vicinity of original data.

Algorithm 3 DivAug

Input: input image x; model f_θ; all possible operations $\mathcal{O}$=$\{$`Sharpness`,$\cdots$, `Contrast`$\}$
Parameters:the number of augmented images per input image E; the number of selected augmented images per input image used for training S
Output:$\mathcal{S}$:= a set of S augmented images of input image x
for $j = 1, \cdots, E$ **do**
    Sample operations $\mathrm{op}_1, \mathrm{op}_2 \sim \mathcal{O}$ uniformly at random
    $p_1 \sim \mathrm{Uniform}(0, 1); p_2 \sim \mathrm{Uniform}(0, 1)$
    $m_1 \sim \mathrm{Uniform}(0, 1); m_2 \sim \mathrm{Uniform}(0, 1)$
    Get sub-policy $t_j := \mathrm{op}_1(\cdot\,; p_1, m_1) \circ \mathrm{op}_2(\cdot\,; p_2, m_2)$
    Generate $x^{t_j} = t(x)$
    Compute $\hat{p}(y|x^{t_j})$= `Softmax`$(f_\theta(x^{t_j}))$
end for
Generate a set of augmented images $\mathcal{S}$ of size S, which is a random subset of $\{x^{t_j}, j = 1, \cdots, E\}$, using k-means++ seeding algorithm on $\{\hat{p}(y|x^{t_j}) : j = 1, \cdots, E\}$
Return$\mathcal{S}$

5.2.3.2 Design of DivAug

According to the definition of Variance Diversity and Equation (5.3), the increase of Variance Diversity directly strengthens the regularization effect of augmented data. Based on this insight, our DivAug framework generates a set of diversely augmented data and minimizes the loss over them. Specifically, DivAug consists of two stages: the sampling stage and the training stage. At the sampling stage, for each original data x_i, we first randomly generate a set of sub-policies $\{t_j\}_{j=1}^m$, where $\{x_i^{t_j}\}_{j=1}^m$ are the set of augmented data $x_i^{t_j}$ corresponding to t_j. Then we sample a subset of augmented data $\mathcal{S}_i \subset \{x_i^{t_j}\}_{j=1}^m$, where $|\mathcal{S}_i| = S < m$. The second stage is the training stage, where we feed the sampled augmented data to the model. Our DivAug framework is illustrated in Figure 5.1. Formally, with the notations introduced in Section 5.2.2 and Section 5.2.3.1, given a neural

network f_θ, we minimize the following objective:

$$\min_{\theta} \quad \frac{1}{N}\sum_{i=1}^{N}[\frac{1}{S}\sum_{x_i^t \in \mathcal{S}_i} l(\hat{p}(y|x_i^t), y)], \tag{5.5}$$

$$\text{s.t.} \quad \mathcal{S}_i = \underset{\substack{\mathcal{S}_i \subset \{x_i^{t_j}\}_{j=1}^{m}, \\ |\mathcal{S}_i|=S}}{\operatorname{argmax}} \mathbb{E}_{x_i^{t_j} \in \mathcal{S}_i} \Delta_i^\top \Delta_i. \tag{5.6}$$

where $\Delta_i = \hat{p}(y|x_i^{t_j}) - \mathbb{E}_{x_i^{t_j} \in \mathcal{S}_i} \hat{p}(y|x_i^{t_j})$. From Equation (5.6), we target at selecting a subset of augmented data $\mathcal{S}_i$, whose corresponded probability vectors have maximum variance. Unfortunately, getting the solution of Equation (5.6) poses a significant computational hurdle. Instead of computing the optimal solution, we efficiently sample $\mathcal{S}_i$ with the k-means++ seeding algorithm [120], which is originally made to generate a good initialization for k-means clustering. k-means++ seeding selects centroids by iteratively sampling points in proportion to their squared distances from the closest centroid that has been chosen. Here, we define the distance between a pair of probability vector as their Euclidean distance. Therefore, k-means++ samples a subset of augmented data where their probability vectors are far apart from each other, which practically leads to a large Variance Diversity. For more details, the k-means++ seeding algorithm is shown in Algorithm 4 in the Appendix C. We show the algorithm of DivAug in Algorithm 3 and remark that the operation is randomly generated. Meanwhile, the two hyperparameters S and E do not need to be tuned on proxy tasks and can be chosen according to available computation resources. Similar to RandAugment, DivAug is a sampling-based method that does not require a separate search process. Note that there is no label information involved in Algorithm 3, which means DivAug is suitable for both semi-supervised learning and supervised learning.

5.3 Experiments

Our experiments aim to answer the following research questions:

- **RQ1**. What is the effect of Variance Diversity on model generalization?
- **RQ2**. How effective is the proposed DivAug compared with other automated data augmenta-

tion methods under the supervised settings?

- **RQ3**. How well does DivAug improve the performance of semi-supervised learning algorithms?

5.3.1 Experimental Settings

Below, we first introduce the datasets and the default augmentation method for them. Then, we will introduce the hyperparameter setting of Divaug (S and E in Algorithm 3) , and the baseline methods for comparison.

- **CIFAR-10 & CIFAR-100 [125]:** The training sets of the two datasets are composed of 50,000 colored images with 10 and 100 classes, respectively. Each image in these two datasets is in size of 32×32. For CIFAR datasets, the default augmentation crops the padded image at a random location, and then horizontally flips it with the probability of 0.5. Then, it applies Cutout [121] to randomly select a 16×16 patch of the image, and set the pixels within the selected patch as zeros.

- **ImageNet [126]:** ImageNet includes colored images of 1,000 classes. The training set has roughly 1.2M images, and the validation set has 50,000 images. The default augmentation randomly crops and resizes images to a size of 224×224, and then horizontally flips it with a probability of 0.5. Subsequently, it performs ColorJitter and PCA to the flipped image [110].

For DivAug, we set $E = 8$ and $S = 4$ for the experiments in Section 5.3.2 and 5.3.3, excluding the ImageNet experiment. For ImageNet, we set $E = 4$ and $S = 2$ due to limited resources. For the semi-supervised learning experiment, we set $E = 4$ and $S = 2$. We did not tune these two hyperparameters, and we choose them mainly according to the available GPU memory.

The methods for comparison are as below: We compare Algorithm 3 with AutoAugment (AA) [119], Fast AutoAugment (Fast AA) [112], Population Based Augmentation (PBA) [114], RandAugment (RA) [113] and Adversarial AutoAugment (Adv. AA) [1]. For each image, the augmentation policy proposed by different methods and the default augmentation are applied in sequence.

5.3.2 Correlation Between Variance Diversity and Generalization

To answer **RQ1**, we calculate the Variance Diversity of augmented data generated by AA, Fast AA, RA, the default augmentation introduced in Section 5.3.1, and DivAug[†]. Then, we report the test accuracy of models trained on augmented data generated by different methods.

Because Variance Diversity is an unsupervised, model-specific measure, for a fair comparison, we first train a Wide-ResNet-40-2 model on CIFAR-10 without applying any data augmentation methods. Then we use it as the f_θ in Equation (5.4) to evaluate all different automated data augmentation methods. To verify the correlation between generalization and Variance Diversity, we calculate the Variance Diversity of augmented data as follows: for each image in the training set, an automated augmentation method is used to randomly generate four augmented images. Then we calculate the Variance Diversity of these four images according to Equation (5.4). We report the averaged Variance Diversity over the entire training set in Figure 5.4.

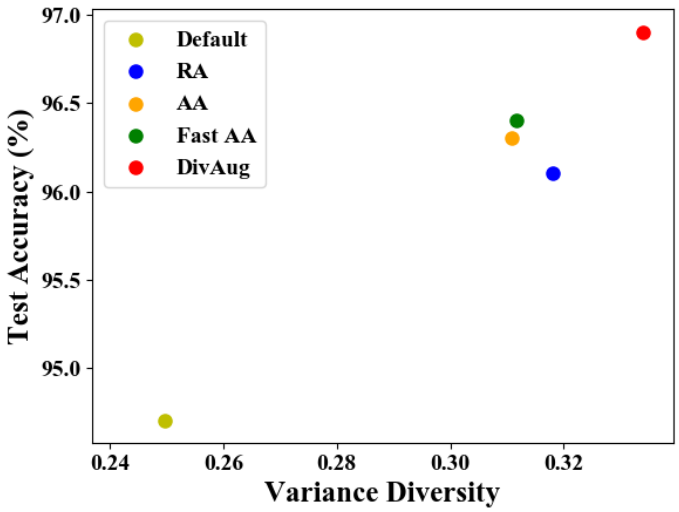


Figure 5.4: The performance gain is positively correlated to Variance Diversity. In general, almost all points lies near the diagonal, and the relative gain in test accuracy increases with larger Variance Diversity.

[†]We do not include Adv. AA because the official code is not released. For PBA, the official code is based on Ray and hard to migrate our codebase for a fair comparison.

Table 5.2: Test accuracy (%) on CIFAR-10 and CIFAR-100. For ImageNet, we report the validation accuracy (%). We compare our method with the default data augmentation (Baseline), AA, Fast AA, PBA, RA, and Adv. AA. Our results are averaged over four trials except ImageNet.

Dataset	Model	Baseline	AA	Fast AA	PBA	RA	Adv. AA	DivAug
CIFAR-10	Wide-ResNet-40-2	94.7	96.3	96.4	-	96.1	-	**96.9**±.1
	Wide-ResNet-28-10	96.1	97.4	97.3	97.4	97.3	**98.1**	**98.1**±.1
	Shake-Shake (26 2x96d)	97.1	98.0	98.0	98.0	98.0	**98.1**	**98.1**±.1
	PyramidNet+ShakeDrop	97.3	98.5	98.3	98.5	98.5	**98.6**	98.5±.1
CIFAR-100	Wide-ResNet-40-2	74.0	79.3	79.4	-	-	-	**81.3**±.3
	Wide-ResNet-28-10	81.2	82.9	82.7	83.3	83.3	**84.5**	84.2±.2
	Shake-Shake (26 2x96d)	82.9	85.7	85.1	84.7	-	**85.9**	85.3±.2
ImageNet	ResNet-50	76.3	77.6	77.6	-	77.6	**79.4**	78.0

Figure 5.4 demonstrates the performance gain and Variance Diversity are positively correlated (the detailed test accuracy is shown in the first row of Table 5.2). As shown in the figure, all automated data augmentation methods could improve the Variance Diversity of augmented data over the default augmentation. Specifically, AA and Fast AA has small Variance Diversity. It makes sense because both of them try to minimize the distribution shift of the augmented data from the original distribution. For example, Fast AA treats the augmented data as the missing point in the training set. As a result, for CIFAR-10, all of the reported sub-policy proposed by AA and Fast AA do not contain the counter-intuitive operation `SamplePair` [112, 119], which limits the Variance Diversity of the augmented data generated by them. In contrast, DivAug has the largest Variance Diversity because it tries to explicitly maximize the Variance Diversity of the augmented data. Notice RA has larger Variance Diversity compared to AA and Fast AA. This might be a result of RA randomly sample operations. As a result, RA samples more distinct sub-policies than AA and Fast AA do and leads to larger diversity. Here we remark that although RA has larger Variance Diversity compared to AA and Fast AA, the model's relative gain in accuracy is smaller compared to those of AA and Fast AA. We provided a detailed analysis in the Appendix D.

5.3.3 The Effectiveness of DivAug Under the Supervised Settings

The main propose of automated data augmentation is to further improve the generalization of models over traditional data augmentation techniques. To answer **RQ2**, we compare our proposed

method with several baselines under the supervised learning settings.

5.3.3.1 Experiment on CIFAR-10 and CIFAR-100

Following [112, 113, 119], we evaluate our proposed method with the following models: Wide-ResNet-28-10, Wide-ResNet-40-2 [127], Shake-Shake (26 2x96d) [128], and Pyramid-Net+ShakeDrop [129, 130]. The details of hyperparameters are shown in Appendix Table E.1.

CIFAR-10 Results: In Table 5.2, we report the test accuracy of these models. For all of these models, our proposed method can achieve better performance compared to previous methods. We achieve 0.7%, 0.8%, 0.7%, 0.8% improvement on Wide-ResNet-28-10 compared to AA, Fast AA, PBA and RA, respectively. Overall, DivAug significantly improves the performances over baselines while achieves comparable performances to those of Adv. AA.

The effect of k-means++ : As illustrated in Section 5.2.1, we remark that RA basically samples sub-policies uniformly in our search space. In contrast, DivAug samples sub-policies using k-means++ seeding algorithm, which pushes the augmented data farther away from each other in the decision space of a given model. Thus, RA can be viewed as the random version of DivAug: the sub-policies picked by RA has an identical percentage of different operations throughout the training process. To understand the effect of k-means++ and how DivAug improves the test accuracy over RA, we further visualize the distribution of sub-policies selected by DivAug with Wide-ResNet-40-2 on CIFAR-10 over the training process. We found that the percentages of some operations picked from the sampled sub-policies, such as `TranslateY`, `ShearY`, `Posterize`, and `SampleParing`, gradually increase along with the training process. In contrast, some color-based operation, such as `Invert`, `Brightness`, `AutoContrast`, and `Color`, gradually decrease along with the training process. In Figure 5.5, we plot the statistics of the two most contrasting operations which exhibit said phenomena, namely, `Posterize` and `Invert`. This behavior is consistent with the discovery that there does not exist an operation beating all other operations throughout the training process [1, 114]. Also, the average probability of applying operations in the selected sub-policies slowly increases with the training process. That means DivAug tends to mildly shift the distribution of augmented images away from the original one

over the training process. From above, it suggests that the sub-policies selected by DivAug evolve throughout the training process.

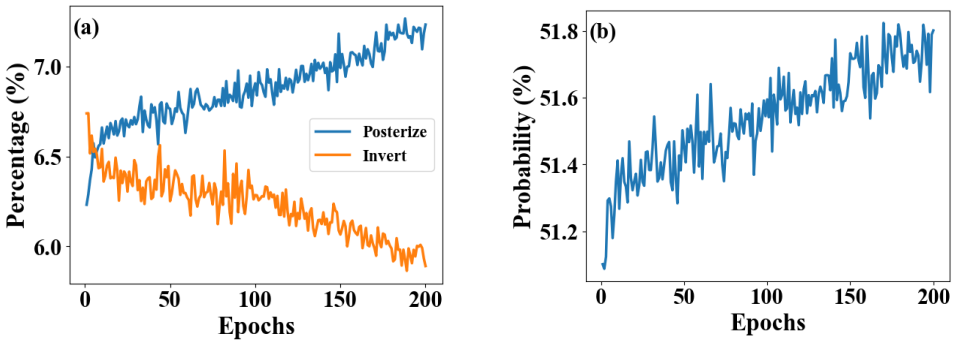


Figure 5.5: The distribution of selected sub-policies evolves along with the training process. For illustration propose, we only plot the statistics of the two most contrasting operations which exhibit said phenomena, namely Posterize and Invert. (a) The statistics of Posterize and Invert in the sub-policies selected by DivAug. (b) The averaged probability of applying operations in the sub-policies selected by DivAug.

Training Efficiency Analysis: DivAug is estimated to be significantly faster than Adv. AA for the following reasons. Following the time cost metric in [118], we estimate the inference cost (see Algorithm 3 line 7) equals half of the training cost. Under the setting of $E = 8$ and $S = 4$, DivAug additionally generates four times more augmented data for training. In contrast, Adv. AA needs to generate eight times more augmented data to achieve the results reported in Table 5.2. Moreover, it also needs a separate phase to search for the best policy. Although the search time for Adv. AA is not reported in [1]. The estimated costs are summarized in Table 5.3.

Table 5.3: Comparison of the total cost of DivAug and Adv. AA on CIFAR-10 relative to RA. The training cost of Adv. AA is cited from [1].

	RA	Adv. AA	DivAug
Training($\times$)	1.0	8.0 + Search Cost	4.5

CIFAR-100 Results: As shown in Table 5.2, DivAug generally achieves non-trivial performance gain over all other methods excluding Adv. AA. However, we note that DivAug does not require label information or a separate search process. Also, DivAug is significantly faster than Adv. AA.

5.3.3.2 Experiment on ImageNet

Following [112, 113, 119], we select ResNet-50 [98] to evaluate our proposed method. The details of the hyperparameters are shown in Appendix Table E.1. As shown in Table 5.2, DivAug outperforms other baselines except Adv. AA. We remark that due to the limited resources, the two hyperparameters in Algorithm 3 are set to $E = 4$ and $S = 2$, respectively. The performance gain from DivAug is expected to be further improved with larger E and S.

5.3.4 The Effectiveness of DivAug Under the Semi-Supervised Setting

One of the key techniques in semi-supervised learning [131] (SSL) is consistency regularization, which encourages the model to produce similar probability vectors when the input data is perturbed by noise. It has been proven that the augmented data produced by state-of-the-art automated methods can serve as a superior source of noise under the consistency regularization framework [132, 133]. Specifically, UDA [132] utilizes RA as the source of perturbation and achieves non-trivial performance gain. Also, it has been theoretically shown that the success of UDA stems from the diversity of augmented data generated by RA [132].

However, most automated data augmentation methods require label information to search for the best policy. Thus, this prerequisite limits their application in SSL. In contrast, our proposed method is suitable for SSL because it is unsupervised and tries to explicitly maximize diversity. This leads to the following question: can SSL benefit from our proposed DivAug (**RQ3**)? To answer this question, following UDA, we change the source of perturbation from RA to DivAug (detailed hyperparameters are shown in the Appendix). Here, we report the averaged results over four trials. As shown in Table 5.4, DivAug can further boost the performance of UDA under different settings. Moreover, the performance gap grows larger when there is less labeled data available. This might

be because, when there is limited labeled data, the regularization effect brought by diversity plays a much bigger role in model performance.

Table 5.4: Error rate (%) comparison with existing methods on CIFAR-10 with 1000, 2000, and 4000 labeled data. All the compared methods use the architecture Wide-ResNet-28-2. For fair comparison, we reproduced the UDA(RA)* result by ourselves using the same codebase.

Methods	CIFAR-10		
	1000	2000	4000
Pseudo-Label[134]	30.91±1.73	21.96±0.42	16.21±0.57
Π-Model[135]	31.53±0.98	23.07±0.66	17.41±0.63
Mean Teacher[136]	17.32±4.00	12.17±0.22	9.19±0.28
MixMatch[137]	7.75±0.32	7.03±0.15	6.42±0.10
UDA(RA)*	7.37±0.15	6.50 ±0.14	5.44±0.15
UDA(DivAug)	**6.94** ±0.12	**6.26**±0.15	**5.40**±0.12

6. CONCLUSIONS AND FUTURE WORK

In this dissertation, we propose a series of methods and frameworks to address different problems in the process of efficient neural architecture search in automated deep learning. In this chapter, we conclude the dissertation and propose the problems to be studied in the future following this dissertation.

6.1 Conclusions

First, we propose a method for learning vector representations from graphs to effectively extract information from the neural architectures. The proposed method, namely, discriminative graph autoencoder (DGA), can learn smooth vector representations for graphs, which also leverages discriminative information based on the graph labels. Specifically, it samples subgraphs from each of the graphs and vectorizes them to feed to the discriminative autoencoder, and then the autoencoder is optimized for two goals: reconstructing the subgraphs and predicting the labels. Experiments on real-world datasets demonstrate that DGA effectively and efficiently learns vector representations from graphs which performed well on classification and visualization tasks.

Second, we propose a novel method for efficient neural architecture search with Bayesian optimization and network morphism. 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 AutoKeras, 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 modern neural architecture search methods.

Third, we propose a new joint neural architecture search and hyperparameter tuning framework including a hyperparameter space and a greedy search algorithm. The algorithm warm starts the search and prefers exploitation over exploration to maximize the efficiency of the joint tuning. It can handle multi-task learning and multi-modal data. The search space is fully customizable.

The method is developed as a new version of AutoKeras with simple APIs to efficiently provide end-to-end deep learning solutions to the users. The model found by AutoKeras can be easily exported and deployed in the production environment with the help of the TensorFlow ecosystem.

Fourth, to address the problem of automated data augmentation, we propose a new measure for quantifying the diversity of augmented data called Variance Diversity by investigating the regularization effect of data augmentation. We validate in experiments that the performance gain from automated data augmentation is highly correlated to Variance Diversity. Based on this measure, we derive the DivAug framework to explicitly maximize Variance Diversity during data augmentation. We demonstrate our proposed method has the practical utility of achieving better performance without the need to search for top policies in a separate phase. Therefore, DivAug can benefit both the supervised tasks and the semi-supervised tasks.

With these methods and frameworks, we successfully address the challenges in the process of efficient neural architecture search for automated deep learning. It covers the neural architecture representation learning, search algorithm, neural architecture evaluations, the joint search space, and some essential parts to be automated in the entire deep learning solution.

6.2 Future Work

In the future, the following open questions may be studied following this dissertation.

First, the AutoML methods may be expanded to more tasks. Most of the existing work on AutoML are using classification and regression tasks as the evaluation of the proposed methods. However, for many tasks, like image segmentation [138], object detection [139, 140], and network analysis [141, 142], task-specific methods may have better performance than general methods.

Second, improving the scalability of the AutoML methods is essential for the adoption of AutoML on real-world problems. For many real-world applications of deep learning, the training dataset can be extremely large, like ImageNet. Running AutoML solutions on such large datasets is not applicable with a single machine. To enable users to use AutoML solutions on large datasets, distributed search and training of the neural network would be a valid approach. How to effectively use the given computational resources for training different models is a non-trivial optimization

problem, which involves many trade-offs. For example, one can start the trials with models with longer training time early so that more trials can be done, or finish some quick trials before the large models so that we may already have some estimations of their performances.

Third, improving the human interaction during the AutoML process would give the user a better experience. Many advanced users of AutoML may want to know more about the AutoML process, for example, what hyperparameters are in the search space, and how are the hyperparameters influencing the performance. They may need this information for their fine-tuning of the model on the final stage before putting it to production. A graphical user interface would be a good solution. How to deliver this information to the users visually is an important problem. It may use various data visualization methods to allow the users to explore this information interactively.

Fourth, considering the various deploy environment of the machine learning models, searching for a model with the user-provided constraints is also an important problem. For example, the user may want to deploy the model in an embedded system, which became a common scenario with the recent advancement in the internet of things (IoT) research. However, the memory resources of such end-devices are usually limited. For some time-sensitive applications, it may also have constraints on the inferencing time of the model. How to find a good model with the provided constraints would be the key to enable much more use cases of AutoML in various deployment environments.

REFERENCES

[1] X. Zhang, Q. Wang, J. Zhang, and Z. Zhong, "Adversarial autoaugment," *arXiv preprint arXiv:1912.11188*, 2019.

[2] T. M. Mitchell, *Machine Learning*. McGraw-Hill, 1997.

[3] Q. Yao, M. Wang, Y. Chen, W. Dai, H. Yi-Qi, L. Yu-Feng, T. Wei-Wei, Y. Qiang, and Y. Yang, "Taking human out of learning applications: A survey on automated machine learning," *arXiv preprint arXiv:1810.13306*, 2018.

[4] F.-F. Li and J. Li, "Cloud automl: Making ai accessible to every business," 2018.

[5] F. Hutter, L. Kotthoff, and J. Vanschoren, *Automatic Machine Learning: Methods, Systems, Challenges*. Springer, 2019.

[6] B. Zoph and Q. V. Le, "Neural architecture search with reinforcement learning," in *International Conference on Learning Representations*, 2016.

[7] L. Li, K. Jamieson, G. DeSalvo, A. Rostamizadeh, and A. Talwalkar, "Hyperband: A novel bandit-based approach to hyperparameter optimization," *The Journal of Machine Learning Research*, 2017.

[8] F. Hutter, H. H. Hoos, and K. Leyton-Brown, "Sequential model-based optimization for general algorithm configuration," in *International Conference on Learning and Intelligent Optimization*, 2011.

[9] J. S. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl, "Algorithms for hyper-parameter optimization," in *Advances in Neural Information Processing Systems*, 2011.

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

[11] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, 1998.

[12] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," in *Advances in neural information processing systems*, 2014.

[13] E. D. Cubuk, B. Zoph, D. Mane, V. Vasudevan, and Q. V. Le, "Autoaugment: Learning augmentation strategies from data," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 113–123, 2019.

[14] A. Narayanan, M. Chandramohan, *et al.*, "subgraph2vec: Learning distributed representations of rooted sub-graphs from large graphs," *arXiv preprint arXiv:1606.08928*, 2016.

[15] T. Kudo, E. Maeda, and Y. Matsumoto, "An application of boosting to graph classification," in *Advances in neural information processing systems*, 2005.

[16] K. Riesen and H. Bunke, *Graph classification and clustering based on vector space embedding*. World Scientific, 2010.

[17] N. Shervashidze and K. M. Borgwardt, "Fast subtree kernels on graphs," in *Advances in neural information processing systems*, 2009.

[18] X. Yan and J. Han, "gspan: Graph-based substructure pattern mining," in *Proceedings of the 2002 SIAM International Conference on Data Mining*, 2002.

[19] N. S. Ketkar, L. B. Holder, and D. J. Cook, "Subdue: Compression-based frequent pattern discovery in graph data," in *Proceedings of the 1st international workshop on open source data mining: frequent pattern mining implementations*, 2005.

[20] X. Yan, H. Cheng, J. Han, and P. S. Yu, "Mining significant graph patterns by leap search," in *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, 2008.

[21] M. Thoma, H. Cheng, A. Gretton, J. Han, H.-P. Kriegel, A. Smola, L. Song, P. S. Yu, X. Yan, and K. Borgwardt, "Near-optimal supervised feature selection among frequent subgraphs," in *Proceedings of the 2009 SIAM International Conference on Data Mining*, 2009.

[22] X. Kong and P. S. Yu, "Semi-supervised feature selection for graph classification," in *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2010.

[23] H. Fröhlich, J. K. Wegner, F. Sieker, and A. Zell, "Optimal assignment kernels for attributed molecular graphs," in *International conference on machine learning*, 2005.

[24] N. Shervashidze, S. Vishwanathan, *et al.*, "Efficient graphlet kernels for large graph comparison," in *Artificial Intelligence and Statistics*, 2009.

[25] N. Shervashidze, P. Schweitzer, *et al.*, "Weisfeiler-lehman graph kernels," *Journal of Machine Learning Research*, 2011.

[26] K. M. Borgwardt and H.-P. Kriegel, "Shortest-path kernels on graphs," in *Proceedings of the 2005 SIAM International Conference on Data Mining*, 2005.

[27] L. Hermansson, F. D. Johansson, and O. Watanabe, "Generalized shortest path kernel on graphs," in *International Conference on Discovery Science*, 2015.

[28] B. L. Douglas, "The weisfeiler-lehman method and graph isomorphism testing," *arXiv preprint arXiv:1101.5211*, 2011.

[29] P. Yanardag and S. Vishwanathan, "A structural smoothing framework for robust graph comparison," in *Advances in neural information processing systems*, 2015.

[30] X. Yan, P. S. Yu, and J. Han, "Graph indexing based on discriminative frequent structure analysis," *TODS*, 2005.

[31] N. Jin, C. Young, and W. Wang, "Gaia: graph classification using evolutionary computation," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, 2010.

[32] X. Kong, W. Fan, and P. S. Yu, "Dual active feature and sample selection for graph classification," in *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2011.

[33] H. Saigo, S. Nowozin, *et al.*, "gboost: a mathematical programming approach to graph classification and regression," *Machine Learning*, 2009.

[34] P. Yanardag and S. Vishwanathan, "Deep graph kernels," in *International Conference on Knowledge Discovery and Data Mining*, 2015.

[35] F. Costa and K. De Grave, "Fast neighborhood subgraph pairwise distance kernel," in *International conference on machine learning*, 2010.

[36] F. Orsini, P. Frasconi, and L. De Raedt, "Graph invariant kernels," in *Proceedings of the*

Twenty-fourth International Joint Conference on Artificial Intelligence, 2015.

[37] M. Neumann, R. Garnett, C. Bauckhage, and K. Kersting, "Propagation kernels: efficient graph kernels from propagated information," *Machine Learning*, 2016.

[38] A. Grover and J. Leskovec, "node2vec: Scalable feature learning for networks," in *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*, 2016.

[39] S. Chang, W. Han, J. Tang, G.-J. Qi, C. C. Aggarwal, and T. S. Huang, "Heterogeneous network embedding via deep architectures," in *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2015.

[40] D. K. Duvenaud, D. Maclaurin, *et al.*, "Convolutional networks on graphs for learning molecular fingerprints," in *Advances in neural information processing systems*, 2015.

[41] F. Scarselli, M. Gori, *et al.*, "The graph neural network model," *IEEE Transactions on Neural Networks*, 2009.

[42] Y. Li, D. Tarlow, M. Brockschmidt, and R. Zemel, "Gated graph sequence neural networks," *arXiv preprint arXiv:1511.05493*, 2015.

[43] M. Niepert, M. Ahmed, and K. Kutzkov, "Learning convolutional neural networks for graphs," in *International conference on machine learning*, 2016.

[44] A. Krizhevsky and G. E. Hinton, "Using very deep autoencoders for content-based image retrieval.," in *ESANN*, 2011.

[45] N. Srivastava, E. Mansimov, and R. Salakhutdinov, "Unsupervised learning of video representations using lstms," in *International conference on machine learning*, 2015.

[46] G. E. Hinton and R. R. Salakhutdinov, "Reducing the dimensionality of data with neural networks," *Science*, 2006.

[47] L. Babai, "Graph isomorphism in quasipolynomial time," in *Proceedings of the forty-eighth annual ACM symposium on Theory of Computing*, 2016.

[48] T. Junttila and P. Kaski, "Engineering an efficient canonical labeling tool for large and sparse graphs," in *2007 Proceedings of the Ninth Workshop on Algorithm Engineering and*

Experiments, 2007.

[49] L. Deng, M. L. Seltzer, *et al.*, "Binary coding of speech spectrograms using a deep auto-encoder," in *INTERSPEECH*, 2010.

[50] B. D. McKay and A. Piperno, "Practical graph isomorphism, ii," *Journal of Symbolic Computation*, 2014.

[51] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, 1997.

[52] A. Debnath, d. C. R. Lopez, *et al.*, "Structure-activity relationship of mutagenic aromatic and heteroaromatic nitro compounds. correlation with molecular orbital energies and hydrophobicity.," *Journal of medicinal chemistry*, 1991.

[53] N. Wale, I. A. Watson, and G. Karypis, "Comparison of descriptor spaces for chemical compound retrieval and classification," *Knowledge and Information Systems*, 2008.

[54] H. Toivonen, A. Srinivasan, R. D. King, S. Kramer, and C. Helma, "Statistical evaluation of the predictive toxicology challenge 2000–2001," *Bioinformatics*, 2003.

[55] K. M. Borgwardt, C. S. Ong, *et al.*, "Protein function prediction via graph kernels," *Bioinformatics*, 2005.

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

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

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

[59] S. Xie, H. Zheng, C. Liu, and L. Lin, "Snas: stochastic neural architecture search," in *International Conference on Learning Representations*, 2019.

[60] R. Luo, F. Tian, T. Qin, E. Chen, and T.-Y. Liu, "Neural architecture optimization," in *Advances in Neural Information Processing Systems*, 2018.

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

[62] H. Cai, L. Zhu, and S. Han, "Proxylessnas: Direct neural architecture search on target task and hardware," in *International Conference on Learning Representations*, 2019.

[63] E. Real, S. Moore, A. Selle, S. Saxena, Y. L. Suematsu, Q. Le, and A. Kurakin, "Large-scale evolution of image classifiers," in *International Conference on Machine Learning*, 2017.

[64] T. Desell, "Large scale evolution of convolutional neural networks using volunteer computing," in *Genetic and Evolutionary Computation Conference Companion*, 2017.

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

[66] M. Suganuma, S. Shirakawa, and T. Nagao, "A genetic programming approach to designing convolutional neural network architectures," in *Genetic and Evolutionary Computation Conference*, 2017.

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

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

[69] H. Cai, T. Chen, W. Zhang, Y. Yu, and J. Wang, "Efficient architecture search by network transformation," in *AAAI Conference on Artificial Intelligence*, 2018.

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

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

[72] T. Wei, C. Wang, Y. Rui, and C. W. Chen, "Network morphism," in *International Conference on Machine Learning*, 2016.

[73] J. Snoek, H. Larochelle, and R. P. Adams, "Practical Bayesian optimization of machine learning algorithms," in *Advances in Neural Information Processing Systems*, 2012.

[74] C. Thornton, F. Hutter, H. H. Hoos, and K. Leyton-Brown, "Auto-WEKA: Combined selection and hyperparameter optimization of classification algorithms," in *International*

Conference on Knowledge Discovery and Data Mining, 2013.

[75] L. Kotthoff, C. Thornton, H. H. Hoos, F. Hutter, and K. Leyton-Brown, "Auto-WEKA 2.0: Automatic model selection and hyperparameter optimization in WEKA," *Journal of Machine Learning Research*, 2016.

[76] M. Feurer, A. Klein, K. Eggensperger, J. Springenberg, M. Blum, and F. Hutter, "Efficient and robust automated machine learning," in *Advances in Neural Information Processing Systems*, 2015.

[77] J. Bergstra, D. Yamins, and D. D. Cox, "Hyperopt: A python library for optimizing the hyperparameters of machine learning algorithms," in *Python in Science Conference*, 2013.

[78] K. Kandasamy, W. Neiswanger, J. Schneider, B. Poczos, and E. Xing, "Neural architecture search with Bayesian optimisation and optimal transport," *Advances in Neural Information Processing Systems*, 2018.

[79] J. Bourgain, "On lipschitz embedding of finite metric spaces in hilbert space," *Israel Journal of Mathematics*, 1985.

[80] Z. Zeng, A. K. Tung, J. Wang, J. Feng, and L. Zhou, "Comparing stars: On approximating graph edit distance," in *International Conference on Very Large Data Bases*, 2009.

[81] H. W. Kuhn, "The hungarian method for the assignment problem," *Naval Research Logistics*, 1955.

[82] B. Haasdonk and C. Bahlmann, "Learning with distance substitution kernels," in *Joint Pattern Recognition Symposium*, 2004.

[83] H. Maehara, "Euclidean embeddings of finite metric spaces," *Discrete Mathematics*, 2013.

[84] P. Auer, N. Cesa-Bianchi, and P. Fischer, "Finite-time analysis of the multiarmed bandit problem," *Machine learning*, 2002.

[85] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE transactions on Systems Science and Cybernetics*, 1968.

[86] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by simulated annealing," *science*, 1983.

[87] F. Chollet *et al.*, "Keras." `https://keras.io`, 2015.

[88] R. S. Olson, N. Bartley, R. J. Urbanowicz, and J. H. Moore, "Evaluation of a tree-based pipeline optimization tool for automating data science," in *Genetic and Evolutionary Computation Conference 2016*, 2016.

[89] Q. Chai and G. Gong, "Verifiable symmetric searchable encryption for semi-honest-but-curious cloud servers," in *International Conference on Communications*, 2012.

[90] L. Buitinck, G. Louppe, M. Blondel, F. Pedregosa, A. Mueller, O. Grisel, V. Niculae, P. Prettenhofer, A. Gramfort, J. Grobler, *et al.*, "API design for machine learning software: experiences from the scikit-learn project," in *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*, 2013.

[91] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, *et al.*, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, 2011.

[92] M. Tan, B. Chen, R. Pang, V. Vasudevan, and Q. V. Le, "Mnasnet: Platform-aware neural architecture search for mobile," *arXiv preprint arXiv:1807.11626*, 2018.

[93] A. Krizhevsky and G. Hinton, "Learning multiple layers of features from tiny images," tech. rep., Citeseer, 2009.

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

[95] L. Kotthoff, C. Thornton, H. Hoos, F. Hutter, and K. Leyton-Brown, "Auto-WEKA 2.0: Automatic model selection and hyperparameter optimization in WEKA," *JMLR*, 2017.

[96] T. O'Malley, E. Bursztein, J. Long, F. Chollet, H. Jin, L. Invernizzi, *et al.*, "Keras Tuner." `https://github.com/keras-team/keras-tuner`, 2019.

[97] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, *et al.*, "TensorFlow: A system for large-scale machine learning," in *OSDI*, 2016.

[98] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *CVPR*, pp. 770–778, 2016.

[99] F. Chollet, "Xception: Deep learning with depthwise separable convolutions," in *CVPR*, 2017.

[100] M. Tan and Q. V. Le, "Efficientnet: Rethinking model scaling for convolutional neural networks," in *ICLR*, 2019.

[101] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," in *NAACL*, 2019.

[102] A. L. Maas, R. E. Daly, P. T. Pham, D. Huang, A. Y. Ng, and C. Potts, "Learning word vectors for sentiment analysis," in *ACL*, 2011.

[103] F. Harrell and T. Cason, "Titanic dataset." `https://www.openml.org/d/40945`, 2017.

[104] R. K. Pace and R. Barry, "Sparse spatial autoregressions," *Statistics & Probability Letters*, 1997.

[105] N. Erickson, J. Mueller, A. Shirkov, H. Zhang, P. Larroy, M. Li, and A. Smola, "AutoGluon-Tabular: Robust and accurate automl for structured data," *arXiv preprint arXiv:2003.06505*, 2020.

[106] L. Wan, M. Zeiler, S. Zhang, Y. Le Cun, and R. Fergus, "Regularization of neural networks using dropconnect," in *ICML*, 2013.

[107] Z. Yang, Z. Dai, Y. Yang, J. Carbonell, R. R. Salakhutdinov, and Q. V. Le, "XLNet: Generalized autoregressive pretraining for language understanding," in *NeurIPS*, 2019.

[108] K. Akylson *et al.*, "Kaggle Titanic dataset leaderboard." `https://www.kaggle.com/c/titanic/leaderboard`, 2020.

[109] H. Wang, "Housing price prediction." `https://www.kaggle.com/harrywang/housing-price-prediction`, 2020.

[110] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, pp. 1097–1105, 2012.

[111] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger, "Densely connected convo-

lutional networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 4700–4708, 2017.

[112] S. Lim, I. Kim, T. Kim, C. Kim, and S. Kim, "Fast autoaugment," in *Advances in Neural Information Processing Systems*, pp. 6665–6675, 2019.

[113] E. D. Cubuk, B. Zoph, J. Shlens, and Q. V. Le, "Randaugment: Practical automated data augmentation with a reduced search space," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops*, pp. 702–703, 2020.

[114] D. Ho, E. Liang, X. Chen, I. Stoica, and P. Abbeel, "Population based augmentation: Efficient learning of augmentation policy schedules," in *International Conference on Machine Learning*, pp. 2731–2741, PMLR, 2019.

[115] R. Hataya, J. Zdenek, K. Yoshizoe, and H. Nakayama, "Faster autoaugment: Learning augmentation strategies using backpropagation," *arXiv preprint arXiv:1911.06987*, 2019.

[116] H. Zhang, M. Cisse, Y. N. Dauphin, and D. Lopez-Paz, "mixup: Beyond empirical risk minimization," *arXiv preprint arXiv:1710.09412*, 2017.

[117] T. Dao, A. Gu, A. J. Ratner, V. Smith, C. De Sa, and C. Ré, "A kernel theory of modern data augmentation," *Proceedings of machine learning research*, vol. 97, p. 1528, 2019.

[118] S. Wu, H. R. Zhang, G. Valiant, and C. Ré, "On the generalization effects of linear transformations in data augmentation," *arXiv preprint arXiv:2005.00695*, 2020.

[119] E. D. Cubuk, B. Zoph, D. Mane, V. Vasudevan, and Q. V. Le, "Autoaugment: Learning augmentation policies from data," *arXiv preprint arXiv:1805.09501*, 2018.

[120] D. Arthur and S. Vassilvitskii, "k-means++ the advantages of careful seeding," in *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, pp. 1027–1035, 2007.

[121] T. DeVries and G. W. Taylor, "Improved regularization of convolutional neural networks with cutout," *arXiv preprint arXiv:1708.04552*, 2017.

[122] H. Inoue, "Data augmentation by pairing samples for images classification," *arXiv preprint arXiv:1801.02929*, 2018.

[123] A. S. Golatkar, A. Achille, and S. Soatto, "Time matters in regularizing deep networks: Weight decay and data augmentation affect early learning dynamics, matter little near convergence," in *Advances in Neural Information Processing Systems*, pp. 10678–10688, 2019.

[124] R. Gontijo-Lopes, S. J. Smullin, E. D. Cubuk, and E. Dyer, "Affinity and diversity: Quantifying mechanisms of data augmentation," *arXiv preprint arXiv:2002.08973*, 2020.

[125] A. Krizhevsky, G. Hinton, *et al.*, "Learning multiple layers of features from tiny images," 2009.

[126] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE conference on computer vision and pattern recognition*, pp. 248–255, Ieee, 2009.

[127] S. Zagoruyko and N. Komodakis, "Wide residual networks," *arXiv preprint arXiv:1605.07146*, 2016.

[128] X. Gastaldi, "Shake-shake regularization," *arXiv preprint arXiv:1705.07485*, 2017.

[129] Y. Yamada, M. Iwamura, T. Akiba, and K. Kise, "Shakedrop regularization for deep residual learning," *IEEE Access*, vol. 7, pp. 186126–186136, 2019.

[130] D. Han, J. Kim, and J. Kim, "Deep pyramidal residual networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 5927–5935, 2017.

[131] O. Chapelle, B. Scholkopf, and A. Zien, "Semi-supervised learning (chapelle, o. et al., eds.; 2006)[book reviews]," *IEEE Transactions on Neural Networks*, vol. 20, no. 3, pp. 542–542, 2009.

[132] Q. Xie, Z. Dai, E. Hovy, M.-T. Luong, and Q. V. Le, "Unsupervised data augmentation for consistency training," *arXiv preprint arXiv:1904.12848*, 2019.

[133] K. Sohn, D. Berthelot, C.-L. Li, Z. Zhang, N. Carlini, E. D. Cubuk, A. Kurakin, H. Zhang, and C. Raffel, "Fixmatch: Simplifying semi-supervised learning with consistency and confidence," *arXiv preprint arXiv:2001.07685*, 2020.

[134] D.-H. Lee, "Pseudo-label: The simple and efficient semi-supervised learning method for deep neural networks," in *Workshop on challenges in representation learning, ICML*, vol. 3,

2013.

[135] S. Laine and T. Aila, "Temporal ensembling for semi-supervised learning," *arXiv preprint arXiv:1610.02242*, 2016.

[136] A. Tarvainen and H. Valpola, "Mean teachers are better role models: Weight-averaged consistency targets improve semi-supervised deep learning results," in *Advances in neural information processing systems*, pp. 1195–1204, 2017.

[137] D. Berthelot, N. Carlini, I. Goodfellow, N. Papernot, A. Oliver, and C. A. Raffel, "Mixmatch: A holistic approach to semi-supervised learning," in *Advances in Neural Information Processing Systems*, pp. 5049–5059, 2019.

[138] C. Liu, L.-C. Chen, F. Schroff, H. Adam, W. Hua, A. Yuille, and L. Fei-Fei, "Auto-deeplab: Hierarchical neural architecture search for semantic image segmentation," *arXiv preprint arXiv:1901.02985*, 2019.

[139] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. C. Berg, "Ssd: Single shot multibox detector," in *European Conference on Computer Vision*, 2016.

[140] G. Ghiasi, T.-Y. Lin, R. Pang, and Q. V. Le, "Nas-fpn: Learning scalable feature pyramid architecture for object detection," *arXiv preprint arXiv:1904.07392*, 2019.

[141] Q. Tan, N. Liu, and X. Hu, "Deep representation learning for social network analysis," *arXiv preprint arXiv:1904.08547*, 2019.

[142] X. Huang, Q. Song, F. Yang, and X. Hu, "Large-scale heterogeneous feature embedding," in *AAAI Conference on Artificial Intelligence*, 2019.

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

APPENDIX A

REPRODUCIBILITY OF NAS EXPERIMENTS

In this section, we provide the details of our implementation and proofs for reproducibility.

- The default architectures used to initialized are introduced.
- The details of the implementation of the four network morphism operations are provided.
- The details of preprocessing the datasets are shown.
- The details of the training process are described.
- The process of using $\rho(\cdot)$ to distort the approximated edit-distance of the neural architectures $d(\cdot, \cdot)$ is introduced.

Notably, the code and detailed documentation are available at AutoKeras official website (`https://autokeras.com`).

A.1 Default Architectures

As we introduced in the experiment section, for all other methods except AK-DP, are using the same three-layer convolutional neural network as the default architecture. The AK-DP is initialized with ResNet, DenseNet, and the three-layer CNN. In the current implementation, ResNet18 and DenseNet121 specifically are chosen as the among all the ResNet and DenseNet architectures.

The three-layer CNN is constructed as follows. Each convolutional layer is actually a convolutional block of a ReLU layer, a batch-normalization layer, the convolutional layer, and a pooling layer. All the convolutional layers are with kernel size equal to three, stride equal to one, and number of filters equal to 64.

All the default architectures share the same fully-connected layers design. After all the convolutional layers, the output tensor passes through a global average pooling layer followed by a dropout

layer, a fully-connected layer of 64 neurons, a ReLU layer, another fully-connected layer, and a softmax layer.

A.2 Network Morphism Implementation

The implementation of the network morphism is introduced from two aspects. First, we describe how the new weights are initialized. Second, we introduce a pool of possible operations which the Bayesian optimizer can select from, *e.g.* the possible start and end points of a skip connection.

The four network morphism operations all involve adding new weights during inserting new layers and expanding existing layers. We initialize the newly added weights with zeros. However, it would create a symmetry prohibiting the newly added weights to learn different values during backpropagation. We follow the Net2Net [71] to add noise to break the symmetry. The amount of noise added is the largest noise possible not changing the output.

There are a large amount of possible network morphism operations we can choose. Although there are only four types of operations we can choose from, a parameter of the operation can be set to a large number of different values. For example, when we use the $deep(G, u)$ operation, we need to choose the location u to insert the layer. In the tree-structured search, we actually cannot exhaust all the operations to get all the children. We will keep sampling from the possible operations until we reach eight children for a node. For the sampling, we randomly sample an operation from $deep$, $wide$ and skip (add and $concat$), with equally likely probability. The parameters of the corresponding operation are sampled accordingly. If it is the $deep$ operation, we need to decide the location to insert the layer. In our implementation, any location except right after a skip-connection. Moreover, we support inserting not only convolutional layers, but activation layers, batch-normalization layers, dropout layer, and fully-connected layers as well. They are randomly sampled with equally likely probability. If it is the $wide$ operation, we need to choose the layer to be widened. It can be any convolutional layer or fully-connected layer, which are randomly sampled with equally likely probability. If it is the skip operations, we need to decide if it is add or $concat$. The start point and end point of a skip-connection can be the output of any layer except the already-exist skip-connection layers. So all the possible skip-connections are generated in the form

of tuples of the start point, end point, and type (add or $concat$), among which we randomly sample a skip-connection with equally likely probability.

A.3 Preprocessing the Datasets

The benchmark datasets, *e.g.*, MNIST, CIFAR10, FASHION, are preprocessed before the neural architecture search. It involves normalization and data augmentation. We normalize the data to the standard normal distribution. For each channel, a mean and a standard deviation are calculated since the values in different channels may have different distributions. The mean and standard deviation are calculated using the training and validation set together. The testing set is normalized using the same values. The data augmentation includes random crop, random horizontal flip, and cutout, which can improve the robustness of the trained model.

A.4 Performance Estimation

During the observation phase, we need to estimate the performance of a neural architecture to update the Gaussian process model in Bayesian optimization. Since the quality of the observed performances of the neural architectures is essential to the neural architecture search algorithm, we propose to train the neural architectures instead of using the performance estimation strategies used in literatures [10, 58, 143]. The quality of the observations is essential to the neural architecture search algorithm. So the neural architectures are 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 criterion in the multi-layer perceptron algorithm in Scikit-Learn [91]. It sets a maximum threshold τ. If the loss of the validation set does not decrease in τ epochs, the training stops. Comparing with the methods using a fixed number of training epochs, it is more adaptive to different neural

architectures.

To avoid being 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 the final metric value on the validation set.

A.5 Distance Distortion

In this section, we introduce how Bourgain theorem is used to distort the learned calculated edit-distance into an isometrically embeddable distance for Euclidean space in the Bayesian optimization process.

From Bourgain theorem, a Bourgain embedding algorithm is designed. The input for the algorithm is a metric distance matrix. Here we use the edit-distance matrix of neural architectures. The outputs of the algorithm are some vectors in Euclidean space corresponding to the instances. In our case, the instances are neural architectures. From these vectors, we can calculate a new distance matrix using Euclidean distance. The objective of calculating these vectors is to minimize the difference between the new distance matrix and the input distance matrix, *i.e.*, minimize the distortions on the distances.

We apply this Bourgain algorithm during the update process of the Bayesian optimization. The edit-distance matrix of previous training examples, *i.e.*, the neural architectures, is stored in memory. Whenever new examples are used to train the Bayesian optimization, the edit-distance is expanded to include the new distances. The distorted distance matrix is computed using Bourgain algorithm from the expanded edit-distance matrix. It is isometrically embeddable to the Euclidean space. The kernel matrix computed using the distorted distance matrix is a valid kernel.

APPENDIX B

PROOF OF THE VALIDITY OF THE KERNEL

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 (3.6), $\forall l$, $w(l) > 0$. $\therefore \forall l_x\ l_y$, $d_l(l_x, l_y) \geq 0$. $\therefore \forall L_x\ L_y$, $D_l(L_x, L_y) \geq 0$. 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 (3.6), 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. $\square$

APPENDIX C

k-means++ SEEDING ALGORITHM

As shown in Algorithm 4, the core idea of k-means++ seeding algorithm is to sample S centers sequentially, where each new center is sampled with probability proportional to the squared distance to its nearest center. The set of centers returned by Algorithm 4 is theoretically guaranteed to far away from each others [120].

Algorithm 4 k-means++seeding Algorithm [120]

Input:$G := \{p_i : p_i \in \mathbb{R}^D\}$; Target size S
Output: Center set C of size S
$C_1 = \{c_1\}$, where c_1 is sampled uniformly at random from G
for $t = 2, \cdots, S$ **do**
    $E_t(x) := \min_{c \in C_{t-1}} ||x - c||_2$
    $c_t \leftarrow$ sample x from G with probability $\frac{E_t^2(x)}{\sum_{x \in G} E_t^2(x)}$
    $C_t \leftarrow C_{t-1} \cup c_t$
end for
ReturnC_S

APPENDIX D

CORRELATION ANALYSIS FOR VARIANCE DIVERSITY

Recently, two measures, Affinity and Diversity, are introduced in [124] for quantifying distribution shift and augmentation diversity, respectively. Across several benchmark datasets and models, it has been observed that the performance gain from data augmentation can be predicted not by either of these alone but by jointly optimizing the two [124]. Specifically, Affinity quantifies how much a sub-policy shifts the training data distribution from the original one. For a set of augmented data, our proposed diversity measure is calculated based on the variance of their probability vectors. Meanwhile, the diversity measure proposed in [124] is defined as the training loss of a given model over the augmented data. Below, we give the formal definition of Affinity and Loss Diversity:

Definition 1 (Affinity [124])**.** *Let D_{train} and D_{val} be training and validation datasets drawn i.i.d. from the same clean data distribution, and let D'_{val} be derived from D_{val} by applying a stochastic augmentation strategy, a, once to each image in D_{val}, $D'_{val} = \{(a(x_i), y) : \forall (x_i, y) \in D_{val}\}$. Further let m be a model trained on D_{train} and $\mathcal{A}(m, D)$ denote the model's accuracy when evaluated on dataset D. The affinity $\tau[a; m; D_{val}]$ is defined as:*

$$\tau[a; m; D_{val}] = \mathcal{A}(m, D'_{val}) - \mathcal{A}(m, D_{val}) \tag{D.1}$$

Definition 2 (Loss Diversity [124])**.** *Let D_{train} be the training set, and D'_{train} be the augmented training set resulting from applying a stochastic augmentation strategy α. For a set of augmented data $\mathcal{S} = \{x'_i\}$, where x'_i is obtained by applying α to x_i, stochastically. Further, given a model m which is trained on D'_{train}, let L'_i be the training loss corresponding to x'_i. The Loss Diversity between $\{x'_i\}$, $\mathcal{D}_{\text{loss}}(\{x'_i\})$, is defined as:*

$$\mathcal{D}_{\text{loss}}(\mathcal{S}) = \mathbb{E}_{x'_i \in \mathcal{S}} L'_i \ ^{*} \tag{D.2}$$

*The original definition of Loss Diversity is defined for the entire training set. To make it comparable to Variance

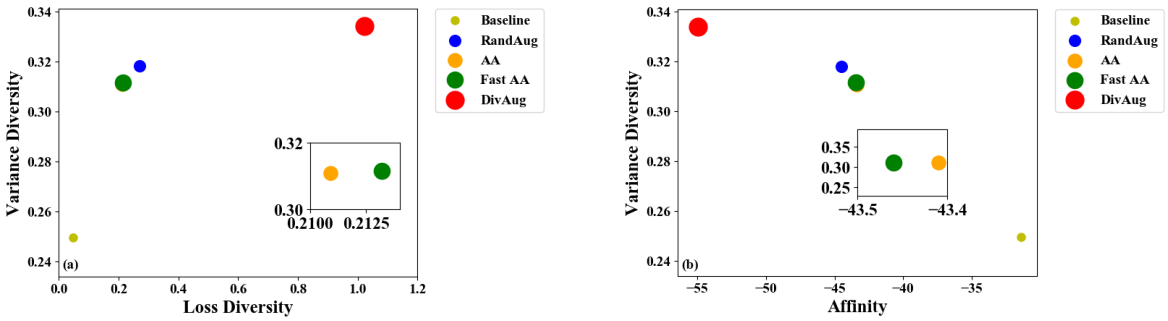


Figure D.1: The performance gain is positively correlated to Variance Diversity. Also, the Loss Diversity and Variance Diversity are highly correlated. The marker size in the legend indicates the relative gain in test accuracy of different methods. (a) The Loss Diversity and the Variance Diversity of augmented data generated by different methods. All points lie near the diagonal of the Figure. In general, the relative gain in test accuracy increases with larger Variance Diversity (b) The Affinity and Variance Diversity of augmented data generated by different methods.

As we analyzed, given a set of augmented data that has large Variance Diversity, it is hard for models to give consistent predictions for them, which will result in a large training loss. Thus, Loss Diversity and Variance Diversity are highly correlated. The main difference between them is that Variance Diversity is an unsupervised measure, i.e., Variance Diversity is not related to the label information.

We further plot the performance gain from each augmentation methods against the Affinity, Loss Diversity, and Variance Diversity of the augmented data generated by them in Figure D.1. In the legend, the marker size indicates the test accuracy of a Wide-ResNet-40-2 model trained with different automated data augmentation methods (The detailed results are shown in the first row of Table 5.2). Figure D.1 demonstrates the Loss Diversity and Variance Diversity are highly correlated, which is consistent with our theoretical analysis. Following [124], we show the Affinity and Variance Diversity of augmented data generated by different methods in Figure D.1 (b). There is a clear trend that the Loss Diversity and Variance Diversity contradict with the Affinity to some extent. We remark that although RA has larger Variance Diversity than AA and Fast AA, the performance gain from RA is smaller. According to the hypothesis in [124], this can be explained

Diversity, we extend the concept to a set of augmented data generated from the same original data x_i.

by RA has smaller Affinity than those of AA and Fast AA. In contrast, although DivAug has the largest Variance Diversity, largest Loss Diversity, and the smallest Affinity, DivAug performs best in terms of test accuracy. We hypothesize that there might exist a sweet spot between Diversity and Affinity, and how to achieve this sweet spot is an interesting future direction for the automated data augmentation methods.

APPENDIX E

DIVAUG EXPERIMENT DETAILS

We list the details of training hyperparameters from the experiments in Section 5.3.3 in Table E.1.

For the semi-supervised learning experiment in Section 5.3.4, we follow the settings in [132] and employ Wide-ResNet-28- 2 [127] as the backbone model and evaluate UDA [132] with varied supervised data sizes. For the experiments on CIFAR-10 with supervised data size 1000, 2000, and 4000, the hyperparameters of them are identical as below: we train the backbone model for 200K steps. We use a batch size of 32 for labeled data and a batch size of 448 for unlabeled data. The softmax temperature τ is set to 0.4. The confidence threshold β is set to 0.8. The backbone model is trained by an SGD optimizer with a learning rate of $1e{-}4$, weight decay of $5e{-}4$, and the Nesterov momentum with the momentum hyperparameter set to 0.9. We remark that all hyperparameters are identical to those reported in [132], except for two differences: we train the backbone model for 200K steps instead of 500K, and we do not apply Exponential Moving Average to the parameters of the backbone model.

Table E.1: Training hyperparameters of CIFAR-10, CIFAR-100 and ImageNet under the supervised settings. LR represents learning rate, and WD represents weight decay. We do not specifically tune these hyperparameters, and all hyperparameters are consistent with those reported in Adversarial AutoAugment [1].

Dataset	Model	Batch Size	LR	WD	Epoch	LR Schedule
CIFAR-10	Wide-ResNet-40-2	128	0.1	5e−4	200	cosine
	Wide-ResNet-28-10	128	0.1	5e−4	200	cosine
	Shake-Shake (26 2x96d)	128	0.2	1e−4	600	cosine
	PyramidNet+ShakeDrop	128	0.1	1e−4	600	cosine
CIFAR-100	Wide-ResNet-40-2	128	0.1	5e−4	200	cosine
	Wide-ResNet-28-10	128	0.1	5e−4	200	cosine
	Shake-Shake (26 2x96d)	128	0.1	5e−4	1200	cosine
ImageNet	ResNet-50	512	0.2	1e−4	120	cosine