Knowledge-Based Systems 212 (2021) 106622

Contents lists available at ScienceDirect

Knowledge-Based Systems

journal homepage: www.elsevier.com/locate/knosys

AutoML: A survey of the state-of-the-art

Xin He, Kaiyong Zhao, Xiaowen Chu *

Department of Computer Science, Hong Kong Baptist University, Hong Kong

ARTICLE INFO

Article history:
Received 8 July 2020
Received in revised form 5 October 2020
Accepted 21 November 2020
Available online 24 November 2020

Keywords:
Deep learning
Automated machine learning (autoML)
Neural architecture search (NAS)
Hyperparameter optimization (HPO)

ABSTRACT

Deep learning (DL) techniques have obtained remarkable achievements on various tasks, such as image recognition, object detection, and language modeling. However, building a high-quality DL system for a specific task highly relies on human expertise, hindering its wide application. Meanwhile, automated machine learning (AutoML) is a promising solution for building a DL system without human assistance and is being extensively studied. This paper presents a comprehensive and up-to-date review of the state-of-the-art (SOTA) in AutoML. According to the DL pipeline, we introduce AutoML methods – covering data preparation, feature engineering, hyperparameter optimization, and neural architecture search (NAS) – with a particular focus on NAS, as it is currently a hot sub-topic of AutoML. We summarize the representative NAS algorithms' performance on the CIFAR-10 and ImageNet datasets and further discuss the following subjects of NAS methods: one/two-stage NAS, one-shot NAS, joint hyperparameter and architecture optimization, and resource-aware NAS. Finally, we discuss some open problems related to the existing AutoML methods for future research.

1. Introduction

In recent years, deep learning has been applied in various fields and used to solve many challenging AI tasks, in areas such as image classification [1,2], object detection [3], and language modeling [4,5]. Specifically, since AlexNet [1] outperformed all other traditional manual methods in the 2012 ImageNet Large Scale Visual Recognition Challenge (ILSVRC) [6], increasingly complex and deep neural networks have been proposed. For example, VGG-16 [7] has more than 130 million parameters, occupies nearly 500 MB of memory space, and requires 15.3 billion floating-point operations to process an image of size 224×224. Notably, however, these models were all manually designed by experts by a trial-and-error process, which means that even experts require substantial resources and time to create well-performing models.

To reduce these onerous development costs, a novel idea of automating the entire pipeline of machine learning (ML) has emerged, i.e., automated machine learning (AutoML). There are various definitions of AutoML. For example, according to [8], AutoML is designed to reduce the demand for data scientists and enable domain experts to automatically build ML applications without much requirement for statistical and ML knowledge. In [9], AutoML is defined as a combination of automation and ML. In a word, AutoML can be understood to involve the automated construction of an ML pipeline on the limited computational budget. With the exponential growth of computing power, AutoML has become a hot topic in both industry and academia. A complete AutoML system can make a dynamic combination of various techniques to form an easy-to-use end-to-end ML pipeline system (as shown in Fig. 1). Many AI companies have created and publicly shared such systems (e.g., Cloud AutoML[1] by Google) to help people with little or no ML knowledge to build high-quality custom models.

As Fig. 1 shows, the AutoML pipeline consists of several processes: data preparation, feature engineering, model generation, and model evaluation. Model generation can be further divided into *search space* and *optimization methods*. The *search space* defines the design principles of ML models, which can be divided into two categories: the traditional ML models (e.g., SVM and KNN), and neural architectures. The optimization methods are classified into *hyperparameter optimization (HPO)* and *architecture optimization (AO)*, where the former indicates the training-related parameters (e.g., the learning rate and batch size), and the latter indicates the model-related parameters (e.g., the number of layer for neural architectures and the number of neighbors for KNN). NAS consists of three important components: the search space of neural architectures, AO methods, and model evaluation methods. AO methods may also refer to *search strategy* [10] or *search policy* [11]. Zoph et al. [12] were one of the first to propose NAS, where a recurrent network is trained by reinforcement learning to automatically search for the best-performing

* Corresponding author.
E-mail addresses: csxinhe@comp.hkbu.edu.hk (X. He), kyzhao@comp.hkbu.edu.hk (K. Zhao), chxw@comp.hkbu.edu.hk (X. Chu).

[1] https://cloud.google.com/automl/.

https://doi.org/10.1016/j.knosys.2020.106622
0950-7051/

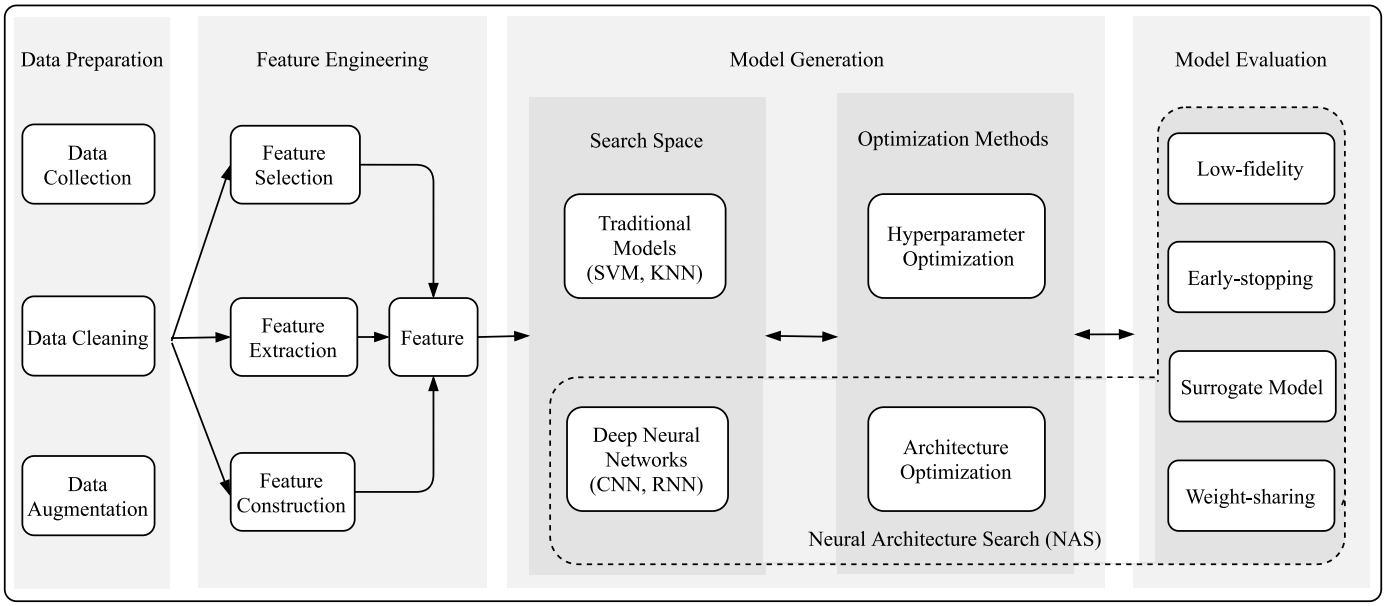


Fig. 1. An overview of AutoML pipeline covering data preparation (Section 2), feature engineering (Section 3), model generation (Section 4) and model evaluation (Section 5).

architecture. Since [12] successfully discovered a neural network achieving comparable results to human-designed models, there has been an explosion of research interest in AutoML, with most focusing on NAS. NAS aims to search for a robust and well-performing neural architecture by selecting and combining different basic operations from a predefined search space. By reviewing NAS methods, we classify the commonly used search space into *entire-structured* [12–14], *cell-based* [13,15–18], *hierarchical* [19] and *morphism-based* [20–22] search space. The commonly used AO methods contain *reinforcement learning* (RL) [12,15,23,16,13], *evolution-based algorithm* (EA) [24–30], and *gradient descent* (GD) [17,31,32], *surrogate model-based optimization* (SMBO) [33–39], and hybrid AO methods [40–44].

Although there are already several excellent AutoML-related surveys [10,45,46,9,8], to the best of our knowledge, our survey covers a broader range of AutoML methods. As summarized in Table 1, [10,45,46] only focus on NAS, while [9,8] cover little of NAS technique. In this paper, we summarize the AutoML-related methods according to the complete AutoML pipeline (Fig. 1), providing beginners with a comprehensive introduction to the field. Notably, many sub-topics of AutoML are large enough to have their own surveys. However, our goal is not to conduct a thorough investigation of all AutoML sub-topics. Instead, we focus on the breadth of research in the field of AutoML. Therefore, we will summarize and discuss some representative methods of each process in the pipeline.

The rest of this paper is organized as follows. The processes of data preparation, feature engineering, model generation, and model evaluation are presented in Sections 2–5. In Section 6, we compare the performance of NAS algorithms on the CIFAR-10 and ImageNet dataset, and discuss several subtopics of great concern in NAS community: one/two-stage NAS, one-shot NAS, joint hyperparameter and architecture optimization, and resource-aware NAS. In Section 7, we describe several open problems in AutoML. We conclude our survey in Section 8.

2. Data preparation

The first step in the ML pipeline is data preparation. Fig. 2 presents the workflow of data preparation, which can be introduced in three aspects: data collection, data cleaning, and data augmentation. Data collection is a necessary step to build a new dataset or extend the existing dataset. The process of data cleaning is used to filter noisy data so that downstream model training is not compromised. Data augmentation plays an important role in enhancing model robustness and improving model performance. The following subsections will cover the three aspects in more detail.

Table 1
Comparison between different AutoML surveys. The "Survey" column gives each survey a label based on their title for increasing the readability. *DP, FE, HPO, NAS* indicate data preparation, feature engineering, hyperparameter optimization and neural architecture search, respectively. "–", "✓", and "†" indicate the content is (1) not mentioned; (2) mentioned detailed; (3) mentioned briefly, in the original paper, respectively.

Survey	DP	FE	HPO	NAS
NAS Survey [10]	–	–	–	✓
A Survey on NAS [45]	–	–	–	✓
NAS Challenges [46]	–	–	–	✓
A Survey on AutoML [9]	–	✓	✓	†
AutoML Challenges [47]	✓	–	✓	†
AutoML Benchmark [8]	✓	✓	✓	–
Ours	✓	✓	✓	✓

2.1. Data collection

ML's deepening study has led to a consensus that high-quality datasets are of critical importance for ML; as a result, numerous open datasets have emerged. In the early stages of ML study, a handwritten digital dataset, i.e., MNIST [48], was developed. After that, several larger datasets like CIFAR-10 and CIFAR-100 [49] and ImageNet [50] were developed. A variety of datasets can also be retrieved by entering the keywords into these websites: Kaggle,[2] Google Dataset Search (GOODS),[3] and Elsevier Data Search.[4]

However, it is usually challenging to find a proper dataset through the above approaches for some particular tasks, such as those related to medical care or other privacy matters. Two types

[2] https://www.kaggle.com.
[3] https://datasetsearch.research.google.com/.
[4] https://www.datasearch.elsevier.com/.

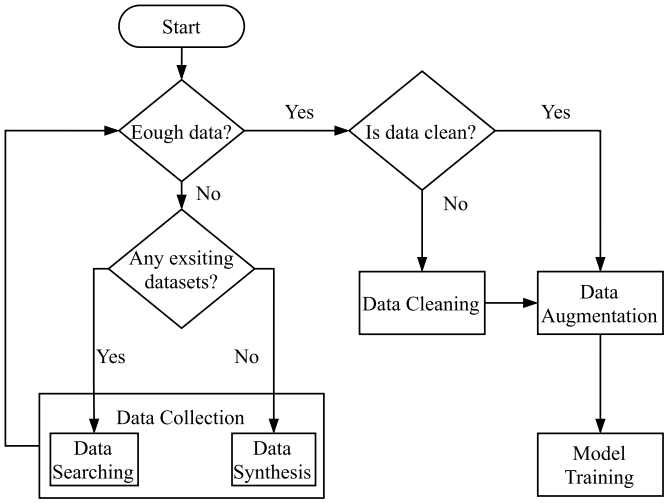


Fig. 2. The flow chart for data preparation.

of methods are proposed to solve this problem: data searching and data synthesis.

2.1.1. Data searching

As the Internet is an inexhaustible data source, searching for Web data is an intuitive way to collect a dataset [51–54]. However, there are some problems with using Web data.

First, the search results may not exactly match the keywords. Thus, unrelated data must be filtered. For example, Krause et al. [55] separate inaccurate results as cross-domain or cross-category noise, and remove any images that appear in search results for more than one category. Vo et al. [56] re-rank relevant results and provide search results linearly, according to keywords.

Second, Web data may be incorrectly labeled or even unlabeled. A learning-based self-labeling method is often used to solve this problem. For example, the active learning method [57] selects the most "uncertain" unlabeled individual examples for labeling by a human, and then iteratively labels the remaining data. Roh et al. [58] provided a review of semi-supervised learning self-labeling methods, which can help take the human out of the loop of labeling to improve efficiency, and can be divided into the following categories: self-training [59,60], co-training [61,62], and co-learning [63]. Moreover, due to the complexity of Web images content, a single label cannot adequately describe an image. Consequently, Yang et al. [51] assigned multiple labels to a Web image, i.e., if the confidence scores of these labels are very close or the label with the highest score is the same as the original label of the image, then this image will be set as a new training sample.

However, the distribution of Web data can be extremely different from that of the target dataset, which will increase the difficulty of training the model. A common solution is to fine-tune these Web data [64,65]. Yang et al. [51] proposed an iterative algorithm for model training and Web data-filtering. Dataset imbalance is another common problem, as some special classes have a very limited number of Web data. To solve this problem, the synthetic minority over-sampling technique (SMOTE) [66] is used to synthesize new minority samples between existing real minority samples, instead of simply up-sampling minority samples or down-sampling the majority samples. In another approach, Guo et al. [67] combined the boosting method with data generation to enhance the generalizability and robustness of the model against imbalanced data sets.

2.1.2. Data synthesis

Data simulator is one of the most commonly used methods to generate data. For some particular tasks, such as autonomous driving, it is not possible to test and adjust a model in the real world during the research phase, due to safety hazards. Therefore, a practical approach to generating data is to use a data simulator that matches the real world as closely as possible. OpenAI Gym [68] is a popular toolkit that provides various simulation environments, in which developers can concentrate on designing their algorithms, instead of struggling to generate data. Wang et al. [69] used a popular game engine, Unreal Engine 4, to build a large synthetic indoor robotics stereo (IRS) dataset, which provides the information for disparity and surface normal estimation. Furthermore, a reinforcement learning-based method is applied in [70] for optimizing the parameters of a data simulator to control the distribution of the synthesized data.

Another novel technique for deriving synthetic data is *Generative Adversarial Networks* (GANs) [71], which can be used to generate images [71–74], tabular [75,76] and text [77] data. Karras et al. [78] applied GAN technique to generate realistic human face images. Oh and Jaroensri et al. [72] built a synthetic dataset, which captures small motion for video-motion magnification. Bowles et al. [74] demonstrated the feasibility of using GAN to generate medical images for brain segmentation tasks. In the case of textual data, applying GAN to text has proved difficult because the commonly used method is to use reinforcement learning to update the gradient of the generator, but the text is discrete, and thus the gradient cannot propagate from discriminator to generator. To solve this problem, Donahue et al. [77] used an autoencoder to encode sentences into a smooth sentence representation to remove the barrier of reinforcement learning. Park et al. [75] applied GAN to synthesize fake tables that are statistically similar to the original table but do not cause information leakage. Similarly, in [76], GAN is applied to generate tabular data like medical or educational records.

2.2. Data cleaning

The collected data inevitably have noise, but the noise can negatively affect the training of the model. Therefore, the process of data cleaning [79,80] must be carried out if necessary. Across the literature, the effort of data cleaning is shifting from crowdsourcing to automation. Traditionally, data cleaning requires specialist knowledge, but access to specialists is limited and generally expensive. Hence, Chu et al. [81] proposed Katara, a knowledge-based and crowd-powered data cleaning system. To improve efficiency, some studies [82,83] proposed only to clean a small subset of the data and maintain comparable results to the case of cleaning the full dataset. However, these methods require a data scientist to design what data cleaning operations are applied to the dataset. BoostClean [84] attempts to automate this process by treating it as a boosting problem. Each data cleaning operation effectively adds a new cleaning operation to the input of the downstream ML model, and through a combination of Boosting and feature selection, a good series of cleaning operations, which can well improve the performance of the ML model, can be generated. AlphaClean [85] transforms data cleaning into a hyperparameter optimization problem, which further increases automation. Specifically, the final data cleaning combinatorial operation in AlphaClean is composed of several pipelined cleaning operations that need to be searched from a predefined search space. Gemp et al. [86] attempted to use meta-learning technique to automate the process of data cleaning.

The data cleaning methods mentioned above are applied to a fixed dataset. However, the real world generates vast amounts of data every day. In other words, how to clean data in a continuous

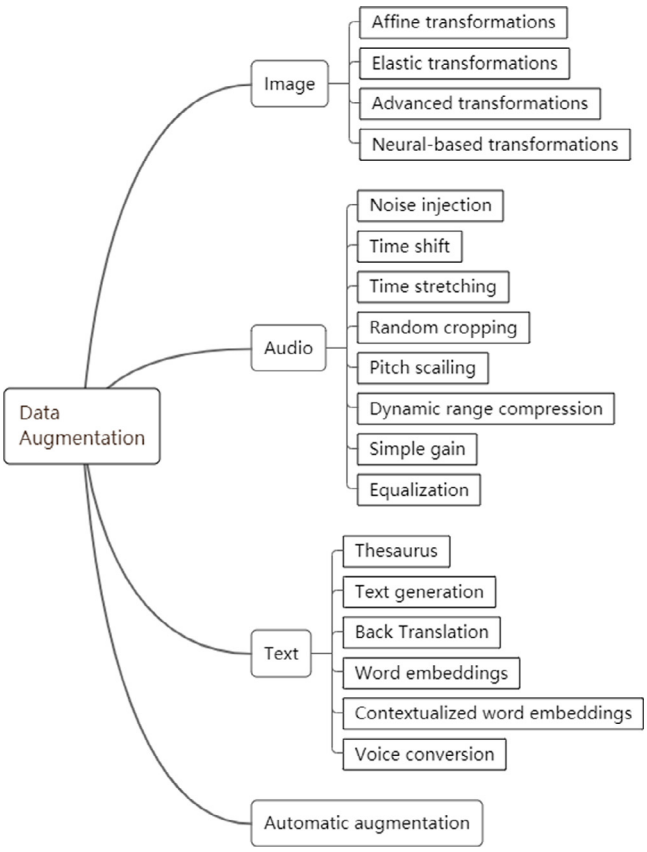


Fig. 3. A classification of data augmentation techniques.

process becomes a worth studying problem, especially for enterprises. Ilyas et al. [87] proposed an effective way of evaluating the algorithms of continuously cleaning data. Mahdavi et al. [88] built a cleaning workflow orchestrator, which can learn from previous cleaning tasks, and proposed promising cleaning workflows for new datasets.

2.3. Data augmentation

To some degree, data augmentation (DA) can also be regarded as a tool for data collection, as it can generate new data based on the existing data. However, DA also serves as a regularizer to avoid over-fitting of model training and has received more and more attention. Therefore, we introduce DA as a separate part of data preparation in detail. Fig. 3 classifies DA techniques from the perspective of data type (image, audio, and text), and incorporates automatic DA techniques that have recently received much attention.

For image data, the affine transformations include rotation, scaling, random cropping, and reflection; the elastic transformations contain the operations like contrast shift, brightness shift, blurring, and channel shuffle; the advanced transformations involve random erasing, image blending, cutout [89], and mixup [90], etc. These three types of common transformations are available in some open source libraries, like torchvision,[5] ImageAug [91], and Albumentations [92]. In terms of neural-based transformations, it can be divided into three categories: adversarial noise [93], neural style transfer [94], and GAN technique [95]. For textual data, Wong et al. [96] proposed two approaches for creating additional training examples: data warping and synthetic over-sampling. The former generates additional samples by applying transformations to data-space, and the latter creates additional samples in feature-space. Textual data can be augmented by synonym insertion or by first translating the text into a foreign language and then translating it back to the original language. In a recent study, Xie et al. [97] proposed a non-domain-specific DA policy that uses noising in RNNs, and this approach works well for the tasks of language modeling and machine translation. Yu et al. [98] proposed a back-translation method for DA to improve reading comprehension. NLPAug [99] is an open-source library that integrates many types of augmentation operations for both textual and audio data.

The above augmentation techniques still require human to select augmentation operations and then form a specific DA policy for specific tasks, which requires much expertise and time. Recently, there are many methods [100–110] proposed to search for augmentation policy for different tasks. AutoAugment [100] is a pioneering work to automate the search for optimal DA policies using reinforcement learning. However, AutoAugment is not efficient as it takes almost 500 GPU hours for one augmentation search. In order to improve search efficiency, a number of improved algorithms have subsequently been proposed using different search strategies, such as gradient descent-based [101,102], Bayesian-based optimization [103], online hyperparameter learning [109], greedy-based search [104] and random search [107]. Besides, LingChen et al. [110] proposed a search-free DA method, namely UniformAugment, by assuming that the augmentation space is approximately distribution invariant.

3. Feature engineering

It is generally accepted that data and features determine the upper bound of ML, and that models and algorithms can only approximate this limit. In this context, feature engineering aims to maximize the extraction of features from raw data for use by algorithms and models. Feature engineering consists of three subtopics: feature selection, feature extraction, and feature construction. Feature extraction and construction are variants of feature transformation, by which a new set of features is created [111]. In most cases, feature extraction aims to reduce the dimensionality of features by applying specific mapping functions, while feature construction is used to expand original feature spaces, and the purpose of feature selection is to reduce feature redundancy by selecting important features. Thus, the essence of automatic feature engineering is, to some degree, a dynamic combination of these three processes.

3.1. Feature selection

Feature selection builds a feature subset based on the original feature set by reducing irrelevant or redundant features. This tends to simplify the model, hence avoiding overfitting and improving model performance. The selected features are usually divergent and highly correlated with object values. According to [112], there are four basic steps in a typical process of feature selection (see Fig. 4), as follows:

The search strategy for feature selection involves three types of algorithms: complete search, heuristic search, and random search. Complete search comprises exhaustive and non-exhaustive searching; the latter can be further split into four methods: breadth-first search, branch and bound search, beam search, and best-first search. Heuristic search comprises sequential forward selection (SFS), sequential backward selection (SBS), and bidirectional search (BS). In SFS and SBS, the features are added from an empty set or removed from a full set, respectively, whereas

[5] https://pytorch.org/docs/stable/torchvision/transforms.html.

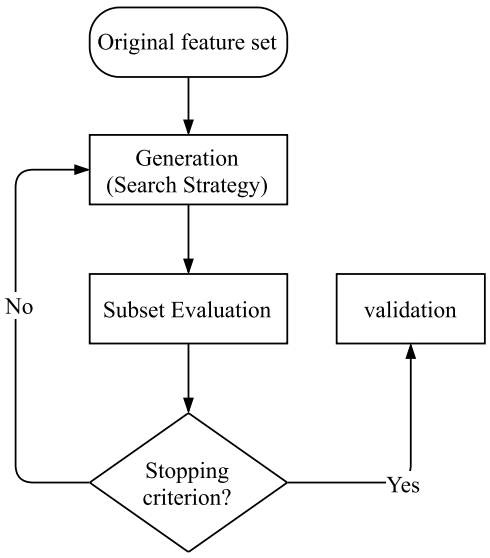


Fig. 4. The iterative process of feature selection. A subset of features is selected based on a search strategy, and then evaluated. The two steps are repeated until the stop criterion is satisfied. After that, a validation procedure is implemented to determine whether the subset is valid.

BS uses both SFS and SBS to search until these two algorithms obtain the same subset. The most commonly used random search methods are simulated annealing (SA) and genetic algorithms (GAs).

Methods of subset evaluation can be divided into three different categories. The first is the filter method, which scores each feature according to its divergence or correlation and then selects features according to a threshold. Commonly used scoring criteria for each feature are variance, the correlation coefficient, the chi-square test, and mutual information. The second is the wrapper method, which classifies the sample set with the selected feature subset, after which the classification accuracy is used as the criterion to measure the quality of the feature subset. The third method is the embedded method, in which variable selection is performed as part of the learning procedure. Regularization, decision tree, and deep learning are all embedded methods.

3.2. Feature construction

Feature construction is a process that constructs new features from the basic feature space or raw data to enhance the robustness and generalizability of the model. Essentially, this is done to increase the representative ability of the original features. This process is traditionally highly dependent on human expertise, and one of the most commonly used methods is preprocessing transformation, such as standardization, normalization, or feature discretization. In addition, the transformation operations for different types of features may vary. For example, operations such as conjunctions, disjunctions and negation are typically used for Boolean features; operations such as minimum, maximum, addition, subtraction, mean are typically used for numerical features, and operations such as Cartesian product [113] and M-of-N [114] are commonly used for nominal features.

It is impossible to manually explore all possibilities. Hence, to further improve efficiency, some automatic feature construction methods [115,114,116,117] have been proposed to automate the process of searching and evaluating the operation combination, and shown to achieve results as good as or superior to those achieved by human expertise. Besides, some feature construction methods, such as decision tree-based methods [115,114] and genetic algorithms [116], require a predefined operation space, while the annotation-based approaches [117] do not, as they can use domain knowledge (in the form of annotation) and the training examples, and hence, can be traced back to the interactive feature-space construction protocol introduced by [118]. Using this protocol, the learner identifies inadequate regions of feature space and, in coordination with a domain expert, adds descriptiveness using existing semantic resources. After selecting possible operations and constructing a new feature, feature-selection techniques are applied to evaluate the new feature.

3.3. Feature extraction

Feature extraction is a dimensionality-reduction process performed via some mapping functions. It extracts informative and non-redundant features according to certain metrics. Unlike feature selection, feature extraction alters the original features. The kernel of feature extraction is a mapping function, which can be implemented in many ways. The most prominent approaches are principal component analysis (PCA), independent component analysis, isomap, nonlinear dimensionality reduction, and linear discriminant analysis (LDA). Recently, the feed-forward neural network approach has become popular; this uses the hidden units of a pretrained model as extracted features. Furthermore, many autoencoder-based algorithms are proposed; for example, Zeng et al. [119] proposed a relation autoencoder model that considers data features and their relationships, while an unsupervised feature-extraction method using autoencoder trees is proposed by [120].

4. Model generation

Model generation is divided into two parts – search space and optimization methods – as shown in Fig. 1. The search space defines the model structures that can be designed and optimized in principle. The types of models can be broadly divided into two categories: traditional ML models, such as support vector machine (SVM) [121] and k-nearest neighbors algorithm (KNN) [122], and deep neural network (DNN). There are two types of parameters for the optimization methods: hyperparameters used for training, such as the learning rate, and those used for model design, such as the filter size and the number of layers for DNN. Neural architecture search (NAS) has recently attracted considerable attention; therefore, in this section, we introduce the search space and optimization methods of NAS technique. Readers who are interested in traditional models (e.g., SVM) can refer to other reviews [9,8].

Fig. 5 presents an overview of the NAS pipeline, which is categorized into the following three dimensions [10,123]: search space, architecture optimization (AO) method,[6] and model evaluation method.

- *Search Space*. The search space defines the design principles of neural architectures. Different scenarios require different search spaces. Here, we summarize four types of commonly used search spaces: entire-structured, cell-based, hierarchical, and morphism-based.
- *Architecture Optimization Method*. The architecture optimization (AO) method defines how to guide the search to efficiently find the model architecture with high performance after the search space is defined.

[6] It can also be referred to as the "search strategy [10,123]", "search policy [11]", or "optimization method [45,9]".

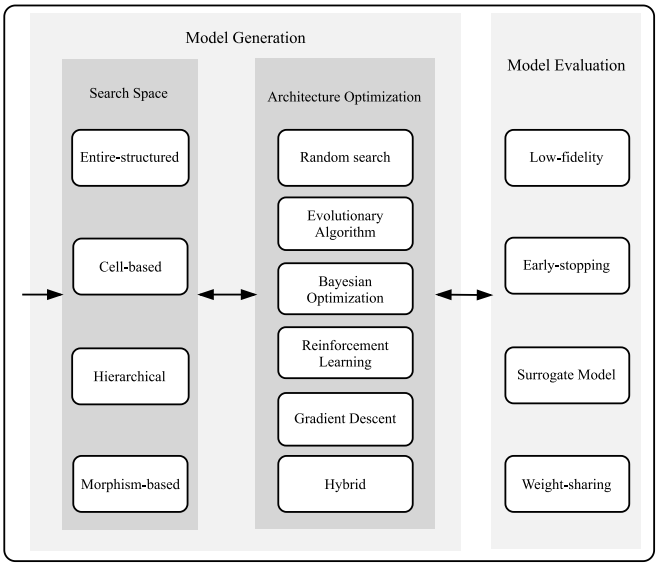


Fig. 5. An overview of neural architecture search pipeline.

- *Model Evaluation Method*. Once a model is generated, its performance needs to be evaluated. The simplest approach of doing this is to train the model to converge on the training set, and then estimate model performance on the validation set; however, this method is time-consuming and resource-intensive. Some advanced methods can accelerate the evaluation process but lose fidelity in the process. Thus, how to balance the efficiency and effectiveness of an evaluation is a problem worth studying.

The search space and AO methods are presented in this section, while the methods of model evaluation are presented in the next section.

4.1. Search space

A neural architecture can be represented as a direct acyclic graph (DAG) comprising B ordered nodes. In DAG, each node and directed edge indicate a feature tensor and an operation, respectively. Eq. (1) presents a formula for computation at any node $Z_k, k \in \{1, 2, \ldots, B\}$.

$$Z_k = \sum_{i=1}^{N_k} o_i(I_i), \quad o_i \in O \tag{1}$$

where N_k indicates the indegree of node Z_k, I_i and o_i represent ith input tensor and its associated operation, respectively, and O is a set of candidate operations, such as convolution, pooling, activation functions, skip connection, concatenation, and addition. To further enhance the model performance, many NAS methods use certain advanced human-designed modules as primitive operations, such as depth-wise separable convolution [124], dilated convolution [125], and squeeze-and-excitation (SE) blocks [126]. The selection and combination of these operations vary with the design of search space. In other words, the search space defines the structural paradigm that AO methods can explore; thus, designing a good search space is a vital but challenging problem. In general, a good search space is expected to exclude human bias and be flexible enough to cover a wider variety of model architectures. Based on the existing NAS studies, we detail the commonly used search spaces as follows.

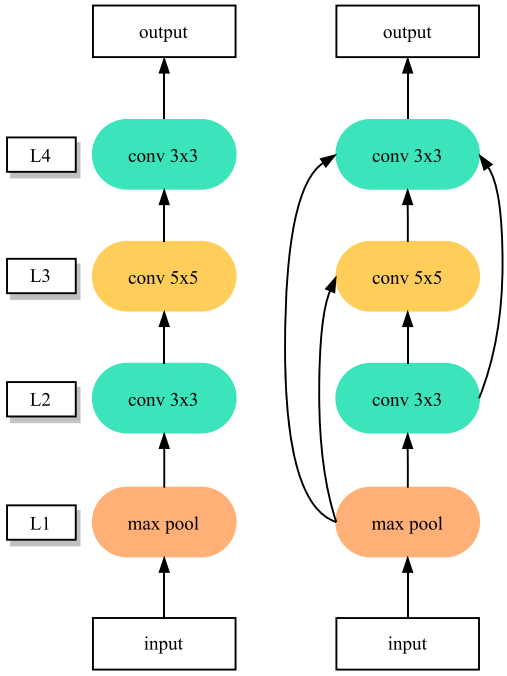


Fig. 6. Two simplified examples of entire-structured neural architectures. Each layer is specified with a different operation, such as convolution and max-pooling operations. The edge indicates the information flow. The skip-connection operation used in the right example can help explore deeper and more complex neural architectures.

4.1.1. Entire-structured search space

The space of entire-structured neural networks [12,13] is one of the most intuitive and straightforward search spaces. Fig. 6 presents two simplified examples of entire-structured models, which are built by stacking a predefined number of nodes, where each node represents a layer and performs a specified operation. The left model shown in Fig. 6 indicates the simplest structure, while the right model is relatively complex, as it permits arbitrary skip connections [2] to exist between the ordered nodes; these connections have been proven effective in practice [12]. Although an entire structure is easy to implement, it has several disadvantages. For example, it is widely accepted that the deeper is the model, the better is its generalization ability; however, searching for such a deep network is onerous and computationally expensive. Furthermore, the generated architecture lacks transferability; that is, a model generated on a small dataset may not fit a larger dataset, which necessitates the generation of a new model for a larger dataset.

4.1.2. Cell-based search space

Motivation. To enable the transferability of the generated model, the cell-based search space has been proposed [15,16,13], in which the neural architecture is composed of a fixed number of repeating cell structures. This design approach is based on the observation that many well-performing human-designed models [2,127] are also built by stacking a fixed number of modules. For example, the ResNet family builds many variants, such as ResNet50, ResNet101, and ResNet152, by stacking several *BottleNeck* modules [2]. Throughout the literature, this repeated module is referred to as a motif, cell, or block, while in this paper, we call it a *cell*.

Design. Fig. 7 (left) presents an example of a final cell-based neural network, which comprises two types of cells: *normal* and *reduction* cells. Thus, the problem of searching for a full neural architecture is simplified into searching for an optimal cell structure

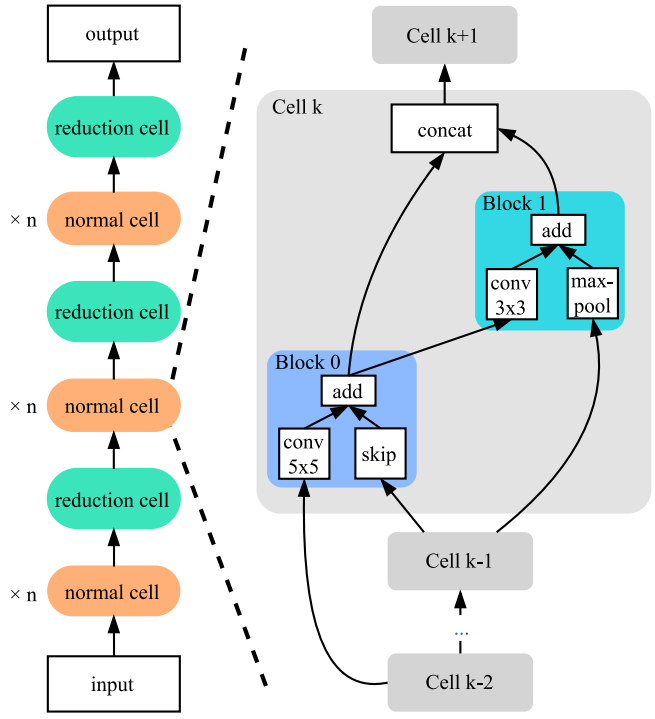


Fig. 7. (Left) Example of a cell-based model comprising three motifs, each with n normal cells and one reduction cell. (Right) Example of a normal cell, which contains two blocks. Each block has two nodes, and each node is specified with different operation and input. The reduction cell is similar in structure to the normal cell.

in the context of cell-based search space. Besides, the output of the normal cell retains the same spatial dimension as the input, and the number of normal cell repeats is usually set manually based on the actual demand. The reduction cell follows behind a normal cell and has a similar structure to that of the normal cell, with the differences being that the width and height of the output feature maps of the reduction cell are half the input, and the number of channels is twice the input. This design approach follows the common practice of manually designing neural networks. Unlike the entire-structured search space, the model built on cell-based search space can be expanded to form a larger model by simply adding more cells without re-searching for the cell structure. Meanwhile, many approaches [17,13,15] have experimentally demonstrated the transferability of the model generated in cell-based search space, such as the model built on CIFAR-10, which can also achieve comparable results to those achieved by SOTA human-designed models on ImageNet.

The design paradigm of the internal cell structure of most NAS studies refers to Zoph et al. [15], who were among the first to propose the exploration of cell-based search space. Fig. 7 (right) shows an example of a normal cell structure. Each cell contains B blocks (here $B = 2$), and each block has two nodes. Each node in a block can be assigned different operations and receive different inputs. The output of two nodes in the block can be combined through addition or concatenation operation. Therefore, each block can be represented by a five-element tuple, (I_1, I_2, O_1, O_2, C), where I_1, I_2 indicate the inputs to the block, while $O_1, O_2 \in \mathcal{O}$ indicate the operations applied to inputs, and C describes how to combine O_1 and O_2. Since the blocks are ordered, the candidate inputs of two nodes in block b_k are selected from the output of the previous two cells and the output of all previous blocks $\{b_i, i < k\}$ within the same cell. For example, the inputs of "Block 1" in Fig. 7 (right) are the outputs of "Block 0" and "Cell k-1".

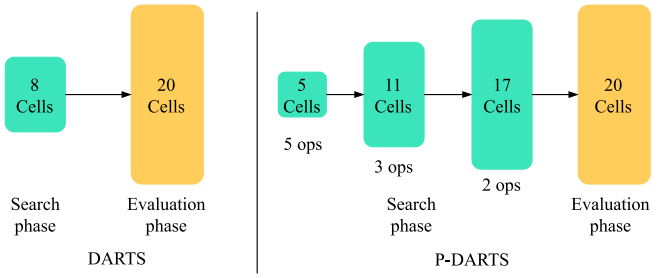


Fig. 8. Difference between DARTS [17] and P-DARTS [128]. Both methods search and evaluate networks on the CIFAR-10 dataset. As the number of cell structures increases from 5 to 11 and then 17, the number of candidate operations is gradually reduced accordingly.

In the actual implementation, certain essential details need to be noted. First, the number of channels may differ for different inputs. A commonly used solution is to apply a calibration operation on each node's input tensor to ensure that all inputs have the same number of channels. The calibration operation generally uses 1×1 convolution filters, such that it will not change the width and height of the input tensor, but keep the channel number of all input tensors consistent. Second, as mentioned above, the input of a node in a block can be obtained from the previous two cells or the previous blocks within the same cell; hence, the cell's output must have the same spatial resolution. To this end, if the input/output resolutions are different, the calibration operation has stride 2; otherwise, it has stride 1. Besides, all blocks have stride 1.

Complexity. Searching for a cell structure is more efficient than searching for an entire structure. To illustrate this, let us assume that there are M predefined candidate operations, the number of layers for both entire and the cell-based structures is L, and the number of blocks in a cell is B. Then, the number of possible entire structures can be expressed as:

$$N_{entire} = M^L \times 2^{\frac{L\times(L-1)}{2}} \tag{2}$$

The number of possible cells is $(M^B \times (B+2)!)^2$. However, as there are two types of cells (i.e., normal and reduction cells), the final size of the cell-based search space is calculated as

$$N_{cell} = (M^B \times (B+2)!)^4 \tag{3}$$

Evidently, the complexity of searching for the entire structure grows exponentially with the number of layers. For an intuitive comparison, we assign the variables in Eqs. (2) and (3) the typical value in the literature, i.e., $M = 5, L = 10, B = 3$; then $N_{entire} = 3.44 \times 10^{20}$ is much larger than $N_{cell} = 5.06 \times 10^{16}$.

Two-stage Gap. The NAS methods of cell-based search space usually comprise two phases: search and evaluation. First, in the search phase, the best-performing model is selected, and then, in the evaluation phase, it is trained from scratch or fine-tuned. However, there exists a large gap in the model depth between the two phases. As Fig. 8 (left) shows, for DARTS [17], the generated model in the search phase only comprises eight cells for reducing the GPU memory consumption, while in the evaluation phase, the number of cells is extended to 20. Although the search phase finds the best cell structure for the shallow model, this does not mean that it is still suitable for the deeper model in the evaluation phase. In other words, simply adding more cells may deteriorate the model performance. To bridge this gap, Chen et al. [128] proposed an improved method based on DARTS, namely progressive-DARTS (P-DARTS), which divides the search phase into multiple stages and gradually increases the depth of the searched networks at the end of each stage,

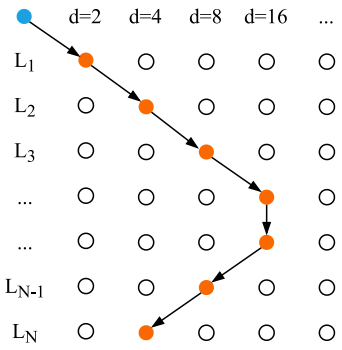


Fig. 9. Network-level search space proposed by [129]. The blue point (top-left) indicates the fixed "stem" structure, the remaining gray and orange points are cell structure, as described above. The black arrows along the orange points indicate the final selected network-level strcuture. "d" and "L" indicate the down sampling rate and layer, respectively. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

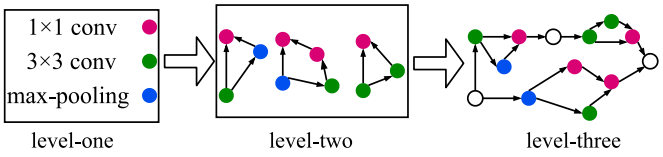


Fig. 10. Example of a three-level hierarchical architecture representation. The level-one primitive operations are assembled into level-two cells. The level-two cells are viewed as primitive operations and assembled into level-three cell. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

hence bridging the gap between search and evaluation. However, increasing the number of cells in the search phase may result in heavier computational overhead. Thus, for reducing the computational consumption, P-DARTS gradually reduces the number of candidate operations from 5 to 3, and then 2, through search space approximation methods, as shown in Fig. 8. Experimentally, P-DARTS obtains a 2.50% error rate on the CIFAR-10 test dataset, outperforming the 2.83% error rate achieved by DARTS.

4.1.3. Hierarchical search space

The cell-based search space enables the transferability of the generated model, and most of the cell-based methods [13,15,23,16,25,26] follow a two-level hierarchy: the inner is the cell level, which selects the operation and connection for each node in the cell, and the outer is the network level, which controls the spatial-resolution changes. However, these approaches focus on the cell level and ignore the network level. As shown in Fig. 7, whenever a fixed number of normal cells are stacked, the spatial dimension of the feature maps is halved by adding a reduction cell. To jointly learn a suitable combination of repeatable cell and network structures, Liu et al. [129] defined a general formulation for a network-level structure, depicted in Fig. 9, from which many existing good network designs can be reproduced. In this way, we can fully explore the different number of channels and sizes of feature maps of each layer in the network.

In terms of the cell level, the number of blocks (B) in a cell is still manually predefined and fixed in the search stage. In other words, B is a new hyperparameter that requires tuning by human input. To address this problem, Liu et al. [19] proposed a novel hierarchical genetic representation scheme, namely *HierNAS*, in which a higher-level cell is generated by iteratively incorporating lower-level cells. As shown in Fig. 10, level-one cells can be some primitive operations, such as 1×1 and 3×3 convolution

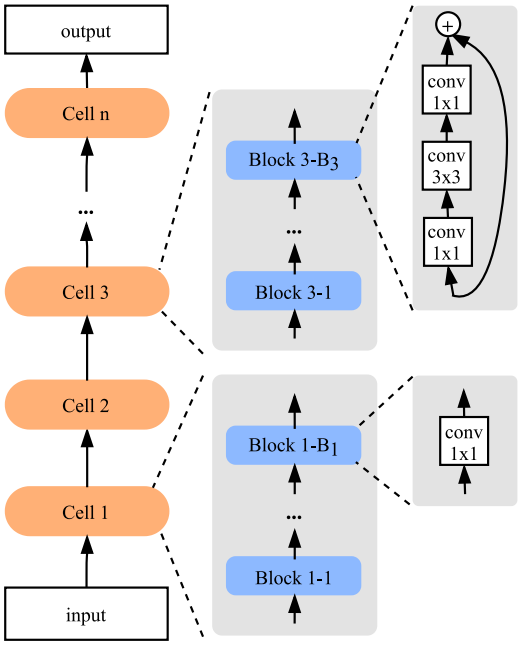


Fig. 11. Factorized hierarchical search space in MnasNet [130]. The final network comprises different cells. Each cell is composed of different number of repeated blocks, and the blocks in different cell are different.

and 3×3 max-pooling, and are the basic components of level-two cells. Then, level-two cells are used as primitive operations to generate level-three cells. The highest-level cell is a single motif corresponding to the full architecture. Besides, a higher-level cell is defined by a learnable adjacency upper-triangular matrix G, where $G_{ij} = k$ indicates that the kth operation 0_k is implemented between nodes i and j. For example, the level-two cell shown in Fig. 10(a) is defined by a matrix G, where $G_{01} = 2, G_{02} = 1, G_{12} = 0$ (the index starts from 0). This method can identify more types of cell structures with more complex and flexible topologies. Similarly, Liu et al. [18] proposed *progressive NAS (PNAS)* to search for the cell progressively, starting from the simplest cell structure, which is composed of only one block, and then expanding to a higher-level cell by adding more possible block structures. Moreover, PNAS improves the search efficiency by using a surrogate model to predict the top-k promising blocks from the search space at each stage of cell construction.

For both HierNAS and PNAS, once a cell structure is searched, it is used in all network layers, which limits the layer diversity. Besides, for achieving both high accuracy and low latency, some studies [130,131] proposed to search for complex and fragmented cell structures. For example, Tan et al. [130] proposed MnasNet, which uses a novel factorized hierarchical search space to generate different cell structures, namely MBConv, for different layers of the final network. Fig. 11 presents the factorized hierarchical search space of MnasNet, which comprises a predefined number of cell structures. Each cell has a different structure and contains a variable number of blocks—whereas all blocks in the same cell exhibit the same structure, those in other cells exhibit different structures. As this design method can achieve a suitable balance between model performance and latency, many subsequent studies [131,132] have referred to it. Owing to the large computational consumption, most of the differentiable NAS (DNAS) techniques (e.g., DARTS) first search for a suitable cell structure on a proxy dataset (e.g., CIFAR10), and then transfer it to a larger target dataset (e.g., ImageNet). Han et al. [132] proposed ProxylessNAS, which can directly search for neural networks on the targeted dataset and hardware platforms by using BinaryConnect [133], which addresses the high memory consumption issue.

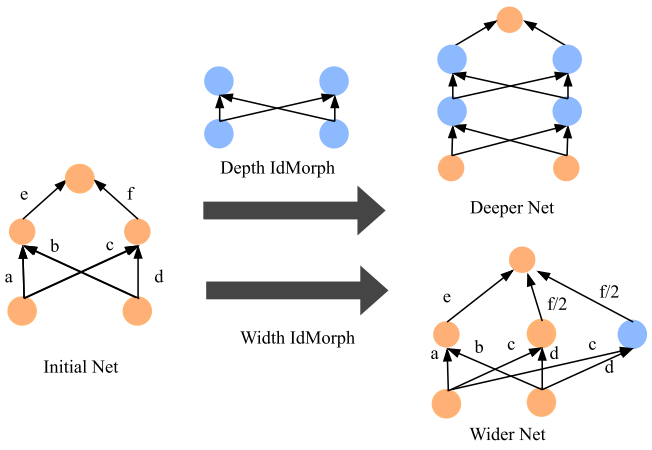


Fig. 12. Net2DeeperNet and Net2WiderNet transformations proposed by [20]. "IdMorph" refers to identity morphism operation. The value on each edge indicates the weight. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

4.1.4. Morphism-based search space

Isaac Newton is reported to have said that "If I have seen further, it is by standing on the shoulders of giants". Similarly, several training tricks have been proposed, such as knowledge distillation [134] and transfer learning [135]. However, these methods do not directly modify the model structure. To this end, Chen et al. [20] proposed the Net2Net technique for designing new neural networks based on an existing network by inserting identity morphism (IdMorph) transformations between the neural network layers. An IdMorph transformation is function-preserving and can be classified into two types – depth and width IdMorph (shown in Fig. 12) – which makes it possible to replace the original model with an equivalent model that is deeper or wider.

However, IdMorph is limited to width and depth changes, and can only modify them separately; moreover, the sparsity of its identity layer can create problems [2]. Therefore, an improved method is proposed, namely network morphism [21], which allows a child network to inherit all knowledge from its well-trained parent network and continue to grow into a more robust network within a shortened training time. Compared with Net2Net, network morphism exhibits the following advantages: (1) it can embed nonidentity layers and handle arbitrary nonlinear activation functions, and (2) it can simultaneously perform depth, width, and kernel size-morphing in a single operation, whereas Net2Net has to separately consider depth and width changes. The experimental results in [21] show that network morphism can substantially accelerate the training process, as it uses one-fifteenth of the training time and achieves better results than the original VGG16.

Several subsequent studies [27,22,136–141] are based on network morphism. For instance, Jin et al. [22] proposed a framework that enables Bayesian optimization to guide the network morphism for an efficient neural architecture search. Wei et al. [136] further improved network morphism at a higher level, i.e., by morphing a convolutional layer into the arbitrary module of a neural network. Additionally, Tan and Le [142] proposed EfficientNet, which re-examines the effect of model scaling on convolutional neural networks, and proved that carefully balancing the network depth, width, and resolution can lead to better performance.

4.2. Architecture optimization

After defining the search space, we need to search for the best-performing architecture, a process we call *architecture optimization (AO)*. Traditionally, the architecture of a neural network is regarded as a set of static hyperparameters that are tuned based on the performance observed on the validation set. However, this process highly depends on human experts and requires considerable time and resources for trial and error. Therefore, many AO methods have been proposed to free humans from this tedious procedure and to search for novel architectures automatically. Below, we detail the commonly used AO methods.

4.2.1. Evolutionary algorithm

The evolutionary algorithm (EA) is a generic population-based metaheuristic optimization algorithm that takes inspiration from biological evolution. Compared with traditional optimization algorithms such as exhaustive methods, EA is a mature global optimization method with high robustness and broad applicability. It can effectively address the complex problems that traditional optimization algorithms struggle to solve, without being limited by the problem's nature.

Encoding Scheme. Different EAs may use different types of encoding schemes for network representation. There are two types of encoding schemes: direct and indirect. Direct encoding is a widely used method that explicitly specifies the phenotype. For example, genetic CNN [30] encodes the network structure into a fixed-length binary string, e.g., 1 indicates that two nodes are connected, and vice versa. Although binary encoding can be performed easily, its computational space is the square of the number of nodes, which is fixed-length, i.e., predefined manually. For representing variable-length neural networks, DAG encoding is a promising solution [28,25,19]. For example, Suganuma et al. [28] used the Cartesian genetic programming (CGP) [143, 144] encoding scheme to represent a neural network built by a list of sub-modules that are defined as DAG. Similarly, in [25], the neural architecture is also encoded as a graph, whose vertices indicate rank-3 tensors or activations (with batch normalization performed with rectified linear units (ReLUs) or plain linear units) and edges indicate identity connections or convolutions. Neuro evolution of augmenting topologies (NEAT) [24,25] also uses a direct encoding scheme, where each node and connection is stored. Indirect encoding specifies a generation rule to build the network and allows for a more compact representation. Cellular encoding (CE) [145] is an example of a system that utilizes indirect encoding of network structures. It encodes a family of neural networks into a set of labeled trees and is based on a simple graph grammar. Some recent studies [146–148,27] have described the use of indirect encoding schemes to represent a network. For example, the network in [27] can be encoded by a function, and each network can be modified using function-preserving network morphism operators. Hence, the child network has increased capacity and is guaranteed to perform at least as well as the parent networks.

Four Steps. A typical EA comprises the following steps: selection, crossover, mutation, and update (Fig. 13):

- **Selection** This step involves selecting a portion of the networks from all generated networks for the crossover, which aims to maintain well-performing neural architectures while eliminating the weak ones. The following three strategies are adopted for network selection. The first is *fitness selection*, in which the probability of a network being selected is proportional to its fitness value, i.e., $P(h_i) = \frac{Fitness(h_i)}{\sum_{j=1}^{N} Fitness(h_j)}$, where h_i indicates the ith network. The second is *rank selection*, which is similar to fitness selection, but

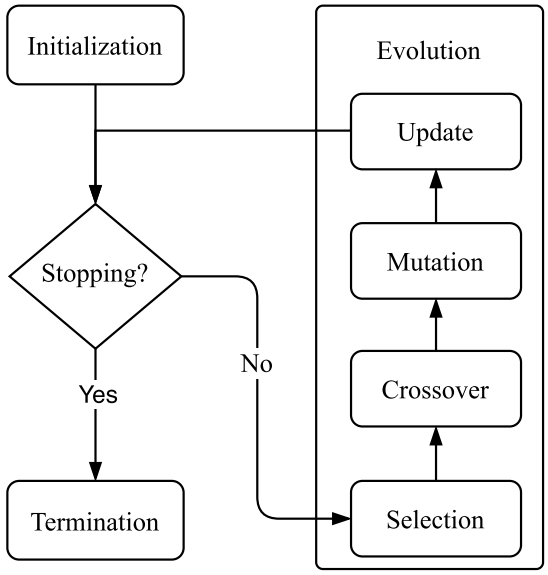


Fig. 13. Overview of the evolutionary algorithm.

with the network's selection probability being proportional to its relative fitness rather than its absolute fitness. The third method is *tournament selection* [25,27,26,19]. Here, in each iteration, k (tournament size) networks are randomly selected from the population and sorted according to their performance; then, the best network is selected with a probability of p, the second-best network has a probability of $p \times (1-p)$, and so on.

- **Crossover** After selection, every two networks are selected to generate a new offspring network, inheriting half of the genetic information of each of its parents. This process is analogous to the genetic recombination, which occurs during biological reproduction and crossover. The particular manner of crossover varies and depends on the encoding scheme. In binary encoding, networks are encoded as a linear string of bits, where each bit represents a unit, such that two parent networks can be combined through one- or multiple-point crossover. However, the crossover of the data arranged in such a fashion can sometimes damage the data. Thus, Xie et al. [30] denoted the basic unit in a crossover as a stage rather than a bit, which is a higher-level structure constructed by a binary string. For cellular encoding, a randomly selected sub-tree is cut from one parent tree to replace a sub-tree cut from the other parent tree. In another approach, NEAT performs an artificial synapsis based on historical markings, adding a new structure without losing track of the gene present throughout the simulation.
- **Mutation** As the genetic information of the parents is copied and inherited by the next generation, gene mutation also occurs. A point mutation [28,30] is one of the most widely used operations and involves randomly and independently flipping each bit. Two types of mutations have been described in [29]: one enables or disables a connection between two layers, and the other adds or removes skip connections between two nodes or layers. Meanwhile, Real and Moore et al. [25] predefined a set of mutation operators, such as altering the learning rate and removing skip connections between the nodes. By analogy with the biological process, although a mutation may appear as a mistake that causes damage to the network structure and leads to a loss of functionality, it also enables the exploration of more novel structures and ensures diversity.

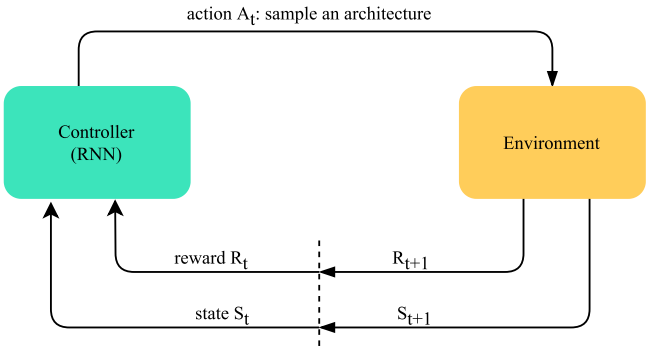


Fig. 14. Overview of neural architecture search using reinforcement learning.

- **Update** Many new networks are generated by completing the above steps, and considering the limitations on computational resources, some of these must be removed. In [25], the worst-performing network of two randomly selected networks is immediately removed from the population. Alternatively, in [26], the oldest networks are removed. Other methods [29,30,28] discard all models at regular intervals. However, Liu et al. [19] did not remove any network from the population, and instead, allowed the network number to grow with time. Zhu et al. [149] regulated the population number through a variable λ, i.e., removed the worst model with probability λ and the oldest model with $1-\lambda$.

4.2.2. Reinforcement learning

Zoph et al. [12] were among the first to apply reinforcement learning (RL) to neural architecture search. Fig. 14 presents an overview of an RL-based NAS algorithm. Here, the controller is usually a recurrent neural network (RNN) that executes an action A_t at each step t to sample a new architecture from the search space and receives an observation of the state S_t together with a reward scalar R_t from the environment to update the controller's sampling strategy. Environment refers to the use of a standard neural network training procedure to train and evaluate the network generated by the controller, after which the corresponding results (such as accuracy) are returned. Many follow-up approaches [23,15,16,13] have used this framework, but with different controller policies and neural-architecture encoding. Zoph et al. [12] first used the policy gradient algorithm [150] to train the controller, and sequentially sampled a string to encode the entire neural architecture. In a subsequent study [15], they used the proximal policy optimization (PPO) algorithm [151] to update the controller, and proposed the method shown in Fig. 15 to build a cell-based neural architecture. MetaQNN [23] is a meta-modeling algorithm using Q-learning with an ϵ-greedy exploration strategy and experience replay to sequentially search for neural architectures.

Although the above RL-based algorithms have achieved SOTA results on the CIFAR-10 and Penn Treebank (PTB) [152] datasets, they incur considerable time and computational resources. For instance, the authors in [12] took 28 days and 800 K40 GPUs to search for the best-performing architecture, and MetaQNN [23] also took 10 days and 10 GPUs to complete its search. To this end, some improved RL-based algorithms have been proposed. BlockQNN [16] uses a distributed asynchronous framework and an early-stop strategy to complete searching on only one GPU within 20 h. The efficient neural architecture search (ENAS) [13] is even better, as it adopts a parameter-sharing strategy in which all child architectures are regarded as sub-graphs of a supernet; this enables these architectures to share parameters, obviating the need to train each child model from scratch. Thus, ENAS

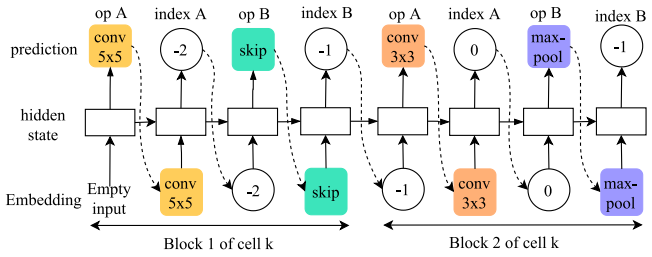


Fig. 15. Example of a controller generating a cell structure. Each block in a cell comprises two nodes, which are specified with different operations and inputs. Indices −2 and −1 indicate the inputs are derived from prev-previous and previous cell, respectively, and the positive index value indicate the index of blocks, e.g., index 0 indicates "Block 1".

took only approximately 10 h using one GPU to search for the best architecture on the CIFAR-10 dataset, which is nearly 1000× faster than [12].

4.2.3. Gradient descent

The above-mentioned search strategies sample neural architectures from a discrete search space. A pioneering algorithm, namely DARTS [17], was among the first gradient descent (GD)-based method to search for neural architectures over a continuous and differentiable search space by using a softmax function to relax the discrete space, as outlined below:

$$\overline{o}_{i,j}(x) = \sum_{k=1}^{K} \frac{\exp\left(\alpha_{i,j}^{k}\right)}{\sum_{l=1}^{K} \exp\left(\alpha_{i,j}^{l}\right)} o^{k}(x) \tag{4}$$

where $o(x)$ indicates the operation performed on input x, $\alpha_{i,j}^{k}$ indicates the weight assigned to the operation o^k between a pair of nodes (i, j), and K is the number of predefined candidate operations. After the relaxation, the task of searching for architectures is transformed into a joint optimization of neural architecture α and the weights of this neural architecture θ. These two types of parameters are optimized alternately, indicating a bilevel optimization problem. Specifically, α and θ are optimized with the validation and the training sets, respectively. The training and the validation losses are denoted by $\mathcal{L}_{train}$ and $\mathcal{L}_{val}$, respectively. Hence, the total loss function can be derived as follows:

$$\begin{aligned} \min_{\alpha} \quad & \mathcal{L}_{val}\left(\theta^{*}, \alpha\right) \\ \text{s.t.} \quad & \theta^{*} = \operatorname{argmin}_{\theta}\ \mathcal{L}_{train}(\theta, \alpha) \end{aligned} \tag{5}$$

Fig. 16 presents an overview of DARTS, where a cell is composed of N (here $N = 4$) ordered nodes and the node z^k (k starts from 0) is connected to the node $z^i, i \in \{k+1, \ldots, N\}$. The operation on each edge $e_{i,j}$ is initially a mixture of candidate operations, each being of equal weight. Therefore, the neural architecture α is a supernet that contains all possible child neural architectures. At the end of the search, the final architecture is derived by retaining only the maximum-weight operation among all mixed operations.

Although DARTS substantially reduces the search time, it incurs several problems. First, as Eq. (5) shows, DARTS describes a joint optimization of the neural architecture and weights as a bilevel optimization problem. However, this problem is difficult to solve directly, because both architecture α and weights θ are high dimensional parameters. Another solution is single-level optimization, which can be formalized as

$$\min_{\theta,\alpha} \mathcal{L}_{\text{train}}(\theta, \alpha) \tag{6}$$

which optimizes both neural architecture and weights together. Although the single-level optimization problem can be efficiently solved as a regular training, the searched architecture α commonly overfits the training set and its performance on the validation set cannot be guaranteed. The authors in [153] proposed mixed-level optimization:

$$\min_{\alpha,\theta} \left[\mathcal{L}_{train}\left(\theta^{*}, \alpha\right) + \lambda \mathcal{L}_{val}\left(\theta^{*}, \alpha\right)\right] \tag{7}$$

where α indicates the neural architecture, θ is the weight assigned to it, and λ is a non-negative regularization variable to control the weights of the training loss and validation loss. When $\lambda = 0$, Eq. (7) reduces to a single-level optimization (Eq. (6)); in contrast, Eq. (7) becomes a bilevel optimization (Eq. (5)). The experimental results presented in [153] showed that mixed-level optimization not only overcomes the overfitting issue of single-level optimization but also avoids the gradient error of bilevel optimization.

Second, in DARTS, the output of each edge is the weighted sum of all candidate operations (shown in Eq. (4)) during the whole search stage, which leads to a linear increase in the requirements of GPU memory with the number of candidate operations. To reduce resource consumption, many subsequent studies [154,155,153,156,131] have developed a differentiable sampler to sample a child architecture from the supernet by using a reparameterization trick, namely Gumbel Softmax [157]. The neural architecture is fully factorized and modeled with a concrete distribution [158], which provides an efficient approach to sampling a child architecture and allows gradient backpropagation. Therefore, Eq. (4) is re-formulated as

$$\overline{o}_{i,j}^{k}(x) = \sum_{k=1}^{K} \frac{\exp\left(\left(\log \alpha_{i,j}^{k} + G_{i,j}^{k}\right)/\tau\right)}{\sum_{l=1}^{K} \exp\left(\left(\log \alpha_{i,j}^{l} + G_{i,j}^{l}\right)/\tau\right)} o^{k}(x) \tag{8}$$

where $G_{i,j}^{k} = -\log(-\log(u_{i,j}^{k}))$ is the kth Gumbel sample, $u_{i,j}^{k}$ is a uniform random variable, and τ is the Softmax temperature. When $\tau \to \infty$, the possibility distribution of all operations between each node pair approximates to one-hot distribution. In GDAS [154], only the operation with the maximum possibility for each edge is selected during the forward pass, while the gradient is backpropagated according to Eq. (8). In other words, only one path of the supernet is selected for training, thereby reducing the GPU memory usage. Besides, ProxylessNAS [132] alleviates the huge resource consumption through path binarization. Specifically, it transforms the real-valued path weights [17] to binary gates, which activates only one path of the mixed operations, and hence, solves the memory issue.

Another problem is the optimization of different operations together, as they may compete with each other, leading to a negative influence. For example, several studies [159,128] have found that skip-connect operation dominates at a later search stage in DARTS, which causes the network to be shallower and leads to a marked deterioration in performance. To solve this problem, DARTS+ [159] uses an additional early-stop criterion, such that when two or more skip-connects occur in a normal cell, the search process stops. In another example, P-DARTS [128] regularizes the search space by executing operation-level dropout to control the proportion of skip-connect operations occurring during training and evaluation.

4.2.4. Surrogate model-based optimization

Another group of architecture optimization methods is surrogate model-based optimization (SMBO) algorithms [33,34,160–166,18,161]. The core concept of SMBO is that it builds a surrogate model of the objective function by iteratively keeping a record of past evaluation results, and uses the surrogate model to predict the most promising architecture. Thus, these methods can substantially shorten the search time and improve efficiency.

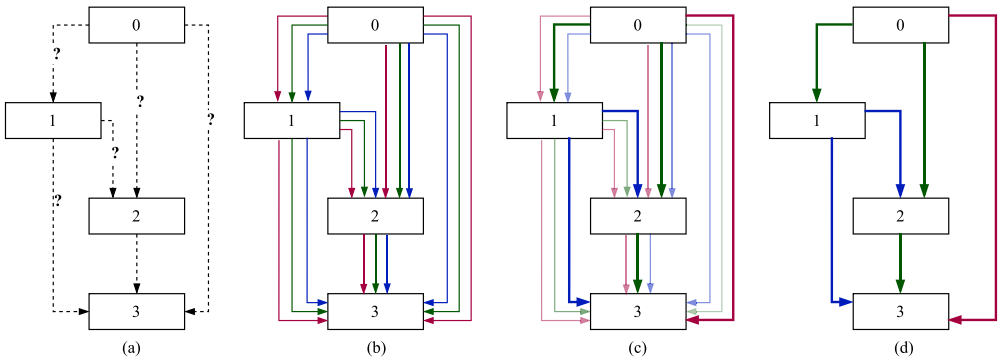


Fig. 16. Overview of DARTS [17]. (a) The data can only flow from lower-level nodes to higher-level nodes, and the operations on edges are initially unknown. (b) The initial operation on each edge is a mixture of candidate operations, each having equal weight. (c) The weight of each operation is learnable and ranges from 0 to 1, but for previous discrete sampling methods, the weight could only be 0 or 1. (d) The final neural architecture is constructed by preserving the maximum weight-value operation on each edge. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

SMBO algorithms differ from the surrogate models, which can be broadly divided into Bayesian optimization (BO) methods (including *Gaussian process (GP)* [167], *random forest (RF)* [37], *tree-structured Parzen estimator (TPE)* [168]), and neural networks [164,169,18,166].

BO [170,171] is one of the most popular methods for hyperparameter optimization. Many recent studies [33,34,160–165] have attempted to apply these SOTA BO methods to AO. For example, in [172,173,160,165,174,175], the validation results of the generated neural architectures were modeled as a Gaussian process, which guides the search for the optimal neural architectures. However, in GP-based BO methods, the inference time scales cubically in the number of observations, and they cannot effectively handle variable-length neural networks. Camero et al. [176] proposed three fixed-length encoding schemes to cope with variable-length problems by using RF as the surrogate model. Similarly, both [33] and [176] used RF as a surrogate model, and [177] showed that it works better in setting high dimensionality than GP-based methods.

Instead of using BO, some studies have used a neural network as the surrogate model. For example, in PNAS [18] and EPNAS [166], an LSTM is derived as the surrogate model to progressively predict variable-sized architectures. Meanwhile, NAO [169] uses a simpler surrogate model, i.e., multilayer perceptron (MLP), and NAO is more efficient and achieves better results on CIFAR-10 than does PNAS [18]. White et al. [164] trained an ensemble of neural networks to predict the mean and variance of the validation results for candidate neural architectures.

4.2.5. Grid and random search

Both grid search (GS) and random search (RS) are simple optimization methods applied to several NAS studies [178–180,11]. For instance, Geifman et al. [179] proposed a modular architecture search space ($\mathcal{A} = \{A(B,i,j)|i \in \{1,2,\ldots,N_{cells}\}, j \in \{1,2,\ldots,N_{blocks}\}\}$) that is spanned by the grid defined by the two corners $A(B,1,1)$ and $A(B,N_{cells},N_{blocks})$, where B is a searched block structure. Evidently, a larger value $N_{cells} \times N_{blocks}$ leads to the exploration of a larger space, but requires more resources.

The authors in [180] conducted an effectiveness comparison between SOTA NAS methods and RS. The results showed that RS is a competitive NAS baseline. Specifically, RS with an early-stopping strategy performs as well as ENAS [13], which is an RL-based leading NAS method. Besides, Yu et al. [11] demonstrated that the SOTA NAS techniques are not significantly better than random search.

4.2.6. Hybrid optimization method

The abovementioned architecture optimization methods have their own advantages and disadvantages. (1) EA is a mature global optimization method with high robustness. However, it requires considerable computational resources [26,25], and its evolution operations (such as crossover and mutations) are performed randomly. (2) Although RL-based methods (e.g., ENAS [13]) can learn complex architectural patterns, the searching efficiency and stability of the RL controller are not guaranteed because it may take several actions to obtain a positive reward. (3) The GD-based methods (e.g., DARTS [17]) substantially improve the searching efficiency by relaxing the categorical candidate operations to continuous variables. Nevertheless, in essence, they all search for a child network from a supernet, which limits the diversity of neural architectures. Therefore, some methods have been proposed to incorporate different optimization methods to capture the best of their advantages; these methods are summarized as follows

EA+RL. Chen et al. [42] integrated reinforced mutations into an EA, which avoids the randomness of evolution and improves the searching efficiency. Another similar method developed in parallel is the evolutionary-neural hybrid controller (Evo-NAS) [41], which also captures the merits of both RL-based methods and EA. The Evo-NAS controller's mutations are guided by an RL-trained neural network, which can explore a vast search space and sample architectures efficiently.

EA+GD. Yang et al. [40] combined the EA and GD-based method. The architectures share parameters within one supernet and are tuned on the training set with a few epochs. Then, the populations and the supernet are directly inherited in the next generation, which substantially accelerates the evolution. The authors in [40] only took 0.4 GPU days for searching, which is more efficient than early EA methods (e.g., AmoebaNet [26] took 3150 GPU days and 450 GPUs for searching).

EA+SMBO. The authors in [43] used RF as a surrogate to predict model performance, which accelerates the fitness evaluation in EA.

GD+SMBO. Unlike DARTS, which learns weights for candidate operations, NAO [169] proposes a variational autoencoder to generate neural architectures and further build a regression model as a surrogate to predict the performance of the generated architecture. The encoder maps the representations of the neural architecture to continuous space, and then a predictor network takes the continuous representations of the neural architecture as input and predicts the corresponding accuracy. Finally, the decoder is used to derive the final architecture from a continuous network representation.

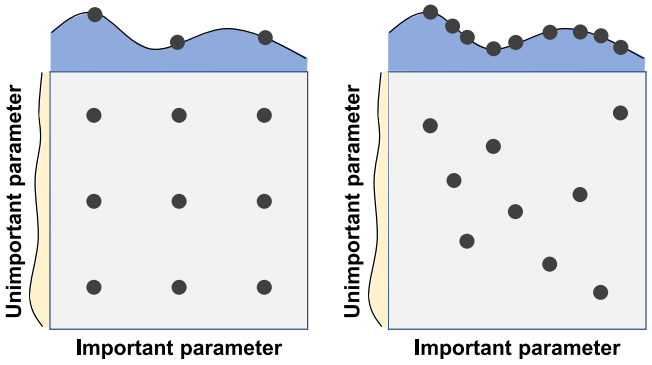


Fig. 17. Examples of grid search (left) and random search (right) in nine trials for optimizing a two-dimensional space function $f(x, y) = g(x) + h(y) \approx g(x)$ [181]. The parameter in $g(x)$ (light-blue part) is relatively important, while that in $h(y)$ (light-yellow part) is not important. In a grid search, nine trials cover only three important parameter values; however, random search can explore nine distinct values of g. Therefore, random search is more likely to find the optimal combination of parameters than grid search. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

4.3. Hyperparameter optimization

Most NAS methods use the same set of hyperparameters for all candidate architectures during the whole search stage; thus, after finding the most promising neural architecture, it is necessary to redesign a hyperparameter set and use it to retrain or fine-tune the architecture. As some HPO methods (such as BO and RS) have also been applied in NAS, we will only briefly introduce these methods here.

4.3.1. Grid and random search

Fig. 17 shows the difference between grid search (GS) and random search (RS): GS divides the search space into regular intervals and selects the best-performing point after evaluating all points; while RS selects the best point from a set of randomly drawn points.

GS is very simple and naturally supports parallel implementation; however, it is computationally expensive and inefficient when the hyperparameter space is very large, as the number of trials grows exponentially with the dimensionality of hyperparameters. To alleviate this problem, Hsu et al. [182] proposed a coarse-to-fine grid search, in which a coarse grid is first inspected to locate a good region, and then a finer grid search is implemented on the identified region. Similarly, Hesterman et al. [183] proposed a contracting GS algorithm, which first computes the likelihood of each point in the grid, and then generates a new grid centered on the maximum-likelihood value. The point separation in the new grid is reduced to half that on the old grid. The above procedure is iterated until the results converge to a local minimum.

Although the authors in [181] empirically and theoretically showed that RS is more practical and efficient than GS, RS does not promise an optimum value. This means that although a longer search increases the probability of finding optimal hyperparameters, it consumes more resources. Li and Jamieson et al. [184] proposed a *hyperband* algorithm to create a tradeoff between the performance of the hyperparameters and resource budgets. The hyperband algorithm allocates limited resources (such as time or CPUs) to only the most promising hyperparameters, by successively discarding the worst half of the configuration settings long before the training process is finished.

4.3.2. BayesIan optimization

Bayesian optimization (BO) is an efficient method for the global optimization of expensive blackbox functions. In this section, we briefly introduce BO. For an in-depth discussion on BO, we recommend readers to refer to the excellent surveys conducted in [171,170,185,186].

BO is an SMBO method that builds a probabilistic model mapping from the hyperparameters to the objective metrics evaluated on the validation set. It well balances exploration (evaluating as many hyperparameter sets as possible) and exploitation (allocating more resources to promising hyperparameters).

Algorithm 1 Surrogate Model-Based Optimization

INPUT: $f, \Theta, S, \mathcal{M}$
$\mathcal{D} \leftarrow$ INITSAMPLES (f, Θ)
for i in $[1, 2, \ldots, T]$ **do**
  $p(y|\theta, \mathcal{D}) \leftarrow$ FITMODEL $(\mathcal{M}, \mathcal{D})$
  $\theta_i \leftarrow \arg\max_{\theta \in \Theta} S(\theta, p(y|\theta, \mathcal{D}))$
  $y_i \leftarrow f(\theta_i)$ ▷ Expensive step
  $\mathcal{D} \leftarrow \mathcal{D} \cup (\theta_i, y_i)$
end for

The steps of SMBO are expressed in Algorithm 1 (adopted from [170]). Here, several inputs need to be predefined initially, including an evaluation function f, search space Θ, acquisition function $\mathcal{S}$, probabilistic model $\mathcal{M}$, and record dataset D. Specifically, D is a dataset that records many sample pairs (θ_i, y_i), where $\theta_i \in \Theta$ indicates a sampled neural architecture and y_i indicates its evaluation result. After the initialization, the SMBO steps are described as follows:

1. The first step is to tune the probabilistic model $\mathcal{M}$ to fit the record dataset D.
2. The acquisition function $\mathcal{S}$ is used to select the next promising neural architecture from the probabilistic model $\mathcal{M}$.
3. The performance of the selected neural architecture is evaluated by f, which is an expensive step as it involves training the neural network on the training set and evaluating it on the validation set.
4. The record dataset D is updated by appending a new pair of results (θ_i, y_i).

The above four steps are repeated T times, where T needs to be specified according to the total time or resources available. The commonly used surrogate models for the BO method are GP, RF, and TPE. Table 2 summarizes the existing open-source BO methods, where GP is one of the most popular surrogate models. However, GP scales cubically with the number of data samples, while RF can natively handle large spaces and scales better to many data samples. Besides, Falkner and Klein et al. [38] proposed the BO-based hyperband (BOHB) algorithm, which combines the strengths of TPE-based BO and hyperband, and hence, performs much better than standard BO methods. Furthermore, FABOLAS [35] is a faster BO procedure, which maps the validation loss and training time as functions of dataset size, i.e., trains a generative model on a sub-dataset that gradually increases in size. Here, FABOLAS is 10–100 times faster than other SOTA BO algorithms and identifies the most promising hyperparameters.

4.3.3. Gradient-based optimization

Another group of HPO methods are *gradient-based optimization (GO)* algorithms [187–192]. Unlike the above blackbox HPO methods (e.g., GS, RS, and BO), GO methods use the gradient information to optimize the hyperparameters and substantially improve the efficiency of HPO. Maclaurin et al. [189] proposed a

Table 2
Open-source Bayesian optimization libraries. GP, RF, and TPE represent *Gaussian process* [167], *random forest* [37], and *tree-structured Parzen estimator* [168], respectively.

Library	Model
Spearmint https://github.com/HIPS/Spearmint	GP
MOE https://github.com/Yelp/MOE	GP
PyBO https://github.com/mwhoffman/pybo	GP
Bayesopt https://github.com/rmcantin/bayesopt	GP
SkGP https://scikit-optimize.github.io	GP
GPyOpt http://sheffieldml.github.io/GPyOpt	GP
SMAC https://github.com/automl/SMAC3	RF
Hyperopt http://hyperopt.github.io/hyperopt	TPE
BOHB https://github.com/automl/HpBandSter	TPE

reversible-dynamics memory-tape approach to handle thousands of hyperparameters efficiently through the gradient information. However, optimizing many hyperparameters is computationally challenging. To alleviate this issue, the authors in [190] used approximate gradient information rather than the true gradient to optimize continuous hyperparameters, where the hyperparameters can be updated before the model is trained to converge. Franceschi et al. [191] studied both reverse- and forward-mode GO methods. The reverse-mode method differs from the method proposed in [189] and does not require reversible dynamics; however, it needs to store the entire training history for computing the gradient with respect to the hyperparameters. The forward-mode method overcomes this problem by using real-time updating hyperparameters, and is demonstrated to significantly improve the efficiency of HPO on large datasets. Chandra [192] proposed a gradient-based ultimate optimizer, which can optimize not only the regular hyperparameters (e.g., learning rate) but also those of the optimizer (e.g., Adam optimizer [193]'s moment coefficient β_1, β_2).

5. Model evaluation

Once a new neural network has been generated, its performance must be evaluated. An intuitive method is to train the network to convergence and then evaluate its performance. However, this method requires extensive time and computing resources. For example, [12] took 800 K40 GPUs and 28 days in total to search. Additionally, NASNet [15] and AmoebaNet [26] required 500 P100 GPUs and 450 K40 GPUs, respectively. In this section, we summarize several algorithms for accelerating the process of model evaluation.

5.1. Low fidelity

As model training time is highly related to the dataset and model size, model evaluation can be accelerated in different ways. First, the number of images or the resolution of images (in terms of image-classification tasks) can be decreased. For example, FABOLAS [35] trains the model on a subset of the training set to accelerate model evaluation. In [194], ImageNet64 $\times$ 64 and its variants 32$\times$32, 16$\times$16 are provided, while these lower resolution datasets can retain characteristics similar to those of the original ImageNet dataset. Second, low-fidelity model evaluation can be realized by reducing the model size, such as by training with fewer filters per layer [15,26]. By analogy to ensemble learning, [195] proposes the *Transfer Series Expansion (TSE)*, which constructs an ensemble estimator by linearly combining a series of basic low-fidelity estimators, hence avoiding the bias that can derive from using a single low-fidelity estimator. Furthermore, Zela et al. [34] empirically demonstrated that there is a weak correlation between performance after short or long training times, thus confirming that a prolonged search for network configurations is unnecessary.

5.2. Weight sharing

In [12], once a network has been evaluated, it is dropped. Hence, the technique of weight sharing is used to accelerate the process of NAS. For example, Wong and Lu et al. [196] proposed transfer neural AutoML, which uses knowledge from prior tasks to accelerate network design. ENAS [13] shares parameters among child networks, leading to a thousand-fold faster network design than [12]. Network morphism based algorithms [20,21] can also inherit the weights of previous architectures, and single-path NAS [197] uses a single-path over-parameterized ConvNet to encode all architectural decisions with shared convolutional kernel parameters.

5.3. Surrogate

The surrogate-based method [198–200,43] is another powerful tool that can approximate the black-box function and predict the performance of neural architectures. In general, once a good approximation has been obtained, it is trivial to find the configurations that directly optimize the original expensive objective. For example, Progressive Neural Architecture Search (PNAS) [18] introduces a surrogate model to guide and accelerate the search process. Although ENAS has been proven to be very efficient, PNAS is even more efficient, as the number of models evaluated by PNAS is over five times that evaluated by ENAS, and PNAS is eight times faster than ENAS in terms of total computational speed. A well-performing surrogate usually requires large amounts of labeled architectures, but the optimization space is too large and hard to quantify, and the evaluation of each configuration is extremely expensive [201]. To alleviate this issue, Luo et al. [202] proposed *SemiNAS*, a semi-supervised NAS method, which leverages amounts of unlabeled architectures to train the surrogate. Initially, the surrogate is only trained with a small number of labeled data pairs *(architecture, accuracy)*, and the surrogate is used to predict the accuracy of the generated architectures without evaluation. New data pairs will be gradually added to the original data to further improve the surrogate.

5.4. Early stopping

Early stopping was first used to prevent overfitting in classical ML, and it has been used in several recent studies [203–205] to accelerate model evaluation by stopping evaluations that are predicted to perform poorly on the validation set. For example, [205] proposes a learning-curve model that is a weighted combination of a set of parametric curve models selected from the literature, thereby enabling the performance of the network to be predicted. Furthermore, [206] presents a novel approach for early stopping based on fast-to-compute local statistics of the computed gradients, which no longer relies on the validation set and allows the optimizer to make full use of all of the training data.

6. NAS discussion

In Section 4, we reviewed the various search space and architecture optimization methods, and in Section 5, we summarized commonly used model evaluation methods. These two sections introduced many NAS studies, which may cause the readers to get lost in details. Therefore, in this section, we summarize and compare these NAS algorithms' performance from a global perspective to provide readers a clearer and more comprehensive understanding of NAS methods' development. Then, we discuss some major topics of the NAS technique.

Table 3
Performance of different NAS algorithms on CIFAR-10. The "AO" column indicates the architecture optimization method. The *dash (–)* indicates that the corresponding information is not provided in the original paper. "c/o" indicates the use of Cutout [89]. RL, EA, GD, RS, and SMBO indicate reinforcement learning, evolution-based algorithm, gradient descent, random search, and surrogate model-based optimization, respectively.

Reference	Published in	#Params (Millions)	Top-1 Acc(%)	GPU Days	#GPUs	AO
ResNet-110 [2]	ECCV16	1.7	93.57	–	–	Manually designed
PyramidNet [207]	CVPR17	26	96.69	–	–	
DenseNet [127]	CVPR17	25.6	96.54	–	–	
GeNet#2 (G-50) [30]	ICCV17	–	92.9	17	–	EA
Large-scale ensemble [25]	ICML17	40.4	95.6	2500	250	
Hierarchical-EAS [19]	ICLR18	15.7	96.25	300	200	
CGP-ResSet [28]	IJCAI18	6.4	94.02	27.4	2	
AmoebaNet-B (N = 6, F = 128)+c/o [26]	AAAI19	34.9	97.87	3150	450 K40	
AmoebaNet-B (N = 6, F = 36)+c/o [26]	AAAI19	2.8	97.45	3150	450 K40	
Lemonade [27]	ICLR19	3.4	97.6	56	8 Titan	
EENA [149]	ICCV19	8.47	97.44	0.65	1 Titan Xp	
EENA (more channels) [149]	ICCV19	54.14	97.79	0.65	1 Titan Xp	
NASv3[12]	ICLR17	7.1	95.53	22,400	800 K40	RL
NASv3+more filters [12]	ICLR17	37.4	96.35	22,400	800 K40	
MetaQNN [23]	ICLR17	–	93.08	100	10	
NASNet-A (7 @ 2304)+c/o [15]	CVPR18	87.6	97.60	2000	500 P100	
NASNet-A (6 @ 768)+c/o [15]	CVPR18	3.3	97.35	2000	500 P100	
Block-QNN-Connection more filter [16]	CVPR18	33.3	97.65	96	32 1080Ti	
Block-QNN-Depthwise, N = 3 [16]	CVPR18	3.3	97.42	96	32 1080Ti	
ENAS+macro [13]	ICML18	38.0	96.13	0.32	1	
ENAS+micro+c/o [13]	ICML18	4.6	97.11	0.45	1	
Path-level EAS [139]	ICML18	5.7	97.01	200	–	
Path-level EAS+c/o [139]	ICML18	5.7	97.51	200	–	
ProxylessNAS-RL+c/o [132]	ICLR19	5.8	97.70	–	–	
FPNAS [208]	ICCV19	5.76	96.99	–	–	
DARTS(first order)+c/o [17]	ICLR19	3.3	97.00	1.5	4 1080Ti	GD
DARTS(second order)+c/o [17]	ICLR19	3.3	97.23	4	4 1080Ti	
sharpDARTS [178]	ArXiv19	3.6	98.07	0.8	1 2080Ti	
P-DARTS+c/o [128]	ICCV19	3.4	97.50	0.3	–	
P-DARTS(large)+c/o [128]	ICCV19	10.5	97.75	0.3	–	
SETN [209]	ICCV19	4.6	97.31	1.8	–	
GDAS+c/o [154]	CVPR19	2.5	97.18	0.17	1	
SNAS+moderate constraint+c/o [155]	ICLR19	2.8	97.15	1.5	1	
BayesNAS [210]	ICML19	3.4	97.59	0.1	1	
ProxylessNAS-GD+c/o [132]	ICLR19	5.7	97.92	–	–	
PC-DARTS+c/o [211]	CVPR20	3.6	97.43	0.1	1 1080Ti	
MiLeNAS [153]	CVPR20	3.87	97.66	0.3	–	
SGAS [212]	CVPR20	3.8	97.61	0.25	1 1080Ti	
GDAS-NSAS [213]	CVPR20	3.54	97.27	0.4	–	
NASBOT [160]	NeurIPS18	–	91.31	1.7	–	SMBO
PNAS [18]	ECCV18	3.2	96.59	225	–	
EPNAS [166]	BMVC18	6.6	96.29	1.8	1	
GHN [214]	ICLR19	5.7	97.16	0.84	–	
NAO+random+c/o [169]	NeurIPS18	10.6	97.52	200	200 V100	RS
SMASH [14]	ICLR18	16	95.97	1.5	–	
Hierarchical-random [19]	ICLR18	15.7	96.09	8	200	
RandomNAS [180]	UAI19	4.3	97.15	2.7	–	
DARTS - random+c/o [17]	ICLR19	3.2	96.71	4	1	
RandomNAS-NSAS [213]	CVPR20	3.08	97.36	0.7	–	
NAO+weight sharing+c/o [169]	NeurIPS18	2.5	97.07	0.3	1 V100	GD+SMBO
RENASNet+c/o [42]	CVPR19	3.5	91.12	1.5	4	EA+RL
CARS [40]	CVPR20	3.6	97.38	0.4	–	EA+GD

6.1. NAS performance comparison

Many NAS studies have proposed several neural architecture variants, where each variant is designed for different scenarios. For instance, some architecture variants perform better but are larger, while some are lightweight for a mobile device but with a performance penalty. Therefore, we only report the representative results of each study. Besides, to ensure a valid comparison, we consider the accuracy and algorithm efficiency as comparison indices. As the number and types of GPUs used vary for different studies, we use *GPU Days* to approximate the efficiency, which is defined as:

$$\text{GPU Days} = N \times D \tag{9}$$

where N represents the number of GPUs, and D represents the actual number of days spent searching.

Tables 3 and 4 present the performances of different NAS methods on CIFAR-10 and ImageNet, respectively. Besides, as most NAS methods first search for the neural architecture based on a small dataset (CIFAR-10), and then transfer the architecture to a larger dataset (ImageNet), the search time for both datasets is the same. The tables show that the early studies on EA- and RL-based NAS methods focused more on high performance, regardless of the resource consumption. For example, although AmoebaNet [26] achieved excellent results for both CIFAR-10 and ImageNet, the searching took 3150 GPU days and 450 GPUs. The subsequent NAS studies attempted to improve the searching efficiency while ensuring the searched model's high performance. For instance, EENA [149] elaborately designs the mutation and crossover operations, which can reuse the learned information to guide the evolution process, and hence, substantially improve

Table 4
Performance of different NAS algorithms on ImageNet. The "AO" column indicates the architecture optimization method. The *dash (–)* indicates that the corresponding information is not provided in the original paper. RL, EA, GD, RS, and SMBO indicate reinforcement learning, evolution-based algorithm, gradient descent, random search, and surrogate model-based optimization, respectively.

Reference	Published in	#Params (Millions)	Top-1/5 Acc(%)	GPU Days	#GPUs	AO
ResNet-152 [2]	CVPR16	230	70.62/95.51	–	–	Manually designed
PyramidNet [207]	CVPR17	116.4	70.8/95.3	–	–	
SENet-154 [126]	CVPR17	–	71.32/95.53	–	–	
DenseNet-201 [127]	CVPR17	76.35	78.54/94.46	–	–	
MobileNetV2 [215]	CVPR18	6.9	74.7/-	–	–	
GeNet#2[30]	ICCV17	–	72.13/90.26	17	–	EA
AmoebaNet-C (N = 4, F = 50) [26]	AAAI19	6.4	75.7/92.4	3150	450 K40	
Hierarchical-EAS [19]	ICLR18	–	79.7/94.8	300	200	
AmoebaNet-C (N = 6, F = 228) [26]	AAAI19	155.3	83.1/96.3	3150	450 K40	
GreedyNAS [216]	CVPR20	6.5	77.1/93.3	1	–	
NASNet-A(4@1056)	ICLR17	5.3	74.0/91.6	2000	500 P100	RL
NASNet-A(6@4032)	ICLR17	88.9	82.7/96.2	2000	500 P100	
Block-QNN [16]	CVPR18	91	81.0/95.42	96	32 1080Ti	
Path-level EAS [139]	ICML18	–	74.6/91.9	8.3	–	
ProxylessNAS(GPU) [132]	ICLR19	–	75.1/92.5	8.3	–	
ProxylessNAS-RL(mobile) [132]	ICLR19	–	74.6/92.2	8.3	–	
MnasNet [130]	CVPR19	5.2	76.7/93.3	1666	–	
EfficientNet-B0[142]	ICML19	5.3	77.3/93.5	–	–	
EfficientNet-B7[142]	ICML19	66	84.4/97.1	–	–	
FPNAS [208]	ICCV19	3.41	73.3/-	0.8	–	
DARTS (searched on CIFAR-10) [17]	ICLR19	4.7	73.3/81.3	4	–	GD
sharpDARTS [178]	Arxiv19	4.9	74.9/92.2	0.8	–	
P-DARTS [128]	ICCV19	4.9	75.6/92.6	0.3	–	
SETN [209]	ICCV19	5.4	74.3/92.0	1.8	–	
GDAS [154]	CVPR19	4.4	72.5/90.9	0.17	1	
SNAS [155]	ICLR19	4.3	72.7/90.8	1.5	–	
ProxylessNAS-G [132]	ICLR19	–	74.2/91.7	–	–	
BayesNAS [210]	ICML19	3.9	73.5/91.1	0.2	1	
FBNet [131]	CVPR19	5.5	74.9/-	216	–	
OFA [217]	ICLR20	7.7	77.3/-	–	–	
AtomNAS [218]	ICLR20	5.9	77.6/93.6	–	–	
MiLeNAS [153]	CVPR20	4.9	75.3/92.4	0.3	–	
DSNAS [219]	CVPR20	–	74.4/91.54	17.5	4 Titan X	
SGAS [212]	CVPR20	5.4	75.9/92.7	0.25	1 1080Ti	
PC-DARTS [211]	CVPR20	5.3	75.8/92.7	3.8	8 V100	
DenseNAS [220]	CVPR20	–	75.3/-	2.7	–	
FBNetV2-L1[221]	CVPR20	–	77.2/-	25	8 V100	
PNAS-5(N = 3, F = 54) [18]	ECCV18	5.1	74.2/91.9	225	–	SMBO
PNAS-5(N = 4, F = 216) [18]	ECCV18	86.1	82.9/96.2	225	–	
GHN [214]	ICLR19	6.1	73.0/91.3	0.84	–	
SemiNAS [202]	CVPR20	6.32	76.5/93.2	4	–	
Hierarchical-random [19]	ICLR18	–	79.6/94.7	8.3	200	RS
OFA-random [217]	CVPR20	7.7	73.8/-	–	–	
RENASNet [42]	CVPR19	5.36	75.7/92.6	–	–	EA+RL
Evo-NAS [41]	Arxiv20	–	75.43/-	740	–	EA+RL
CARS [40]	CVPR20	5.1	75.2/92.5	0.4	–	EA+GD

the efficiency of EA-based NAS methods. ENAS [13] is one of the first RL-based NAS methods to adopt the parameter-sharing strategy, which reduces the number of GPU budgets to 1 and shortens the searching time to less than one day. We also observe that gradient descent-based architecture optimization methods can substantially reduce the computational resource consumption for searching, and achieve SOTA results. Several follow-up studies have been conducted to achieve further improvement and optimization in this direction. Interestingly, RS-based methods can also obtain comparable results. The authors in [180] demonstrated that RS with weight-sharing could outperform a series of powerful methods, such as ENAS [13] and DARTS [17].

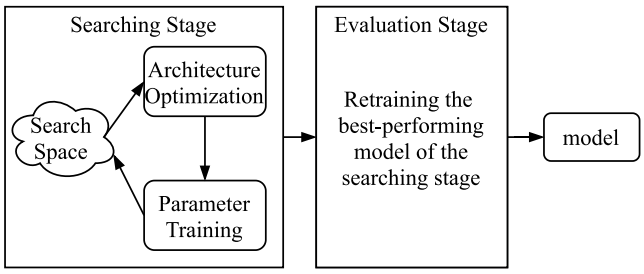
6.1.1. Kendall Tau metric

As RS is comparable to more sophisticated methods (e.g., DARTS and ENAS), a natural question is, what are the advantages and significance of the other AO algorithms compared with RS? Researchers have tried to use other metrics to answer this question, rather than simply considering the model's final accuracy. Most NAS methods comprise two stages: (1) search for a best-performing architecture on the training set and (2) expand it to a deeper architecture and estimate it on the validation set. However, there usually exists a large gap between the two stages. In other words, the architecture that achieves the best result in the training set is not necessarily the best one for the validation set. Therefore, instead of merely considering the final accuracy and search time cost, many NAS studies [219,222,213,11,123] have used Kendall Tau (τ) metric [223] to evaluate the correlation of the model performance between the search and evaluation stages. The parameter τ is defined as

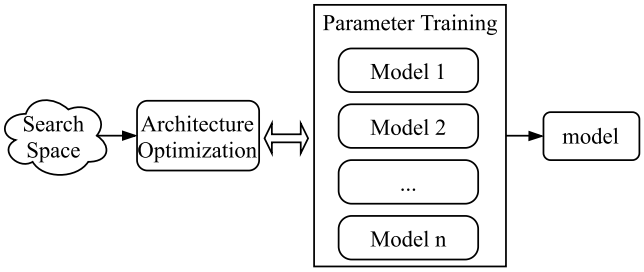
$$\tau = \frac{N_C - N_D}{N_C + N_D} \tag{10}$$

where N_C and N_D indicate the numbers of concordant and discordant pairs. τ is a number in the range [-1, 1] with the following properties:

- $\tau = 1$: two rankings are identical
- $\tau = -1$: two rankings are completely opposite.
- $\tau = 0$: there is no relationship between two rankings.



(a) Two-stage NAS comprises the searching stage and evaluation stage. The best-performing model of the searching stage is further retrained in the evaluation stage.



(b) One-stage NAS can directly deploy a well-performing model without extra retraining or fine-tuning. The two-way arrow indicates that the processes of architecture optimization and parameter training run simultaneously.

Fig. 18. Illustration of two- and one-stage neural architecture search flow.

6.1.2. NAS-bench dataset

Although Tables 3 and 4 present a clear comparison between different NAS methods, the results of different methods are obtained under different settings, such as training-related hyper-parameters (e.g., batch size and training epochs) and data augmentation (e.g., Cutout [89]). In other words, the comparison is not quite fair. In this context, NAS-Bench-101 [224] is a pioneering work for improving the reproducibility. It provides a tabular dataset containing 423,624 unique neural networks generated and evaluated from a fixed graph-based search space and mapped to their trained and evaluated performance on CIFAR-10. Meanwhile, Dong et al. [225] further built NAS-Bench-201, which is an extension to NAS-Bench-101 and has a different search space, results on multiple datasets (CIFAR-10, CIFAR-100, and ImageNet-16-120 [194]), and more diagnostic information. Similarly, Klyuchnikov et al. [226] proposed a NAS-Bench for the NLP task. These datasets enable NAS researchers to focus solely on verifying the effectiveness and efficiency of their AO algorithms, avoiding repetitive training for selected architectures and substantially helping the NAS community to develop.

6.2. One-stage vs. Two-stage

The NAS methods can be roughly divided into two classes according to the flow – two-stage and one-stage – as shown in Fig. 18.

Two-stage NAS comprises the *searching stage* and *evaluation stage*. The *searching stage* involves two processes: architecture optimization, which aims to find the optimal architecture, and parameter training, which is to train the found architecture's parameter. The simplest idea is to train all possible architectures' parameters from scratch and then choose the optimal architecture. However, it is resource-consuming (e.g., NAS-RL [12] took 22,400 GPU days with 800 K40 GPUs for searching)), which is infeasible for most companies and institutes. Therefore, most NAS methods (such as ENAS [13] and DARTS [17]) sample and train many candidate architectures in the searching stage, and then further retrain the best-performing architecture in the evaluation stage.

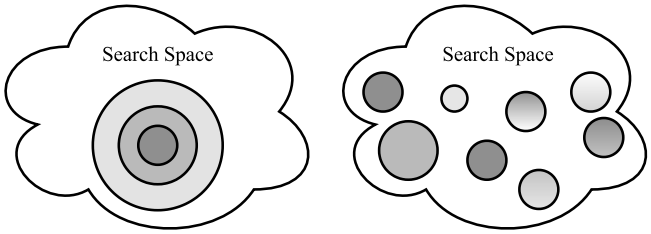


Fig. 19. (Left) One-shot models. (Right) Non-one-shot models. Each circle indicates a different model, and its area indicates the model size. We use concentric circles to represent one-shot models, as they share the weights with each other.

One-stage NAS refers to a class of NAS methods that can export a well-designed and well-trained neural architecture without extra retraining, by running AO and parameter training simultaneously. In this way, the efficiency can be substantially improved. However, model architecture and its weight parameters are highly coupled; it is difficult to optimize them simultaneously. Several recent studies [217,227,228,218] have attempted to overcome this challenge. For instance, the authors in [217] proposed the *progressive shrinking* algorithm to post-process the weights after the training was completed. They first pretrained the entire neural network, and then progressively fine-tuned the smaller networks that shared weights with the complete network. Based on well-designed constraints, the performance of all subnetworks was guaranteed. Thus, given a target deployment device, a specialized subnetwork can be directly exported without fine-tuning. However, [217] was still computational resource-intensive, as the whole process took 1200 GPU hours with V100 GPUs. BigNAS [228] revisited the conventional training techniques of stand-alone networks, and empirically proposed several techniques to handle a wider set of models, ranging in size from 200M to 1G FLOPs, whereas [217] only handled models under 600M FLOPs. Both AtomNAS [218] and DSNAS [219] proposed an end-to-end one-stage NAS framework to further boost the performance and simplify the flow.

6.3. One-shot/weight-sharing

One-shot≠one-stage. Note that one shot is not exactly equivalent to one stage. As mentioned above, we divide the NAS studies into one- and two-stage methods according to the flow (Fig. 18), whereas whether a NAS algorithm belongs to a one-shot method depends on whether the candidate architectures share the same weights (Fig. 19). However, we observe that most one-stage NAS methods are based on the one-shot paradigm.

What is One-shot NAS? One-shot NAS methods embed the search space into an overparameterized supernet, and thus, all possible architectures can be derived from the supernet. Fig. 19 shows the difference between the search spaces of one-shot and non-one-shot NAS. Each circle indicates a different architecture, where the architectures of one-shot NAS methods share the same weights. One-shot NAS methods can be divided into two categories according to how to handle AO and parameter training: coupled and decoupled optimization [229,216].

Coupled optimization. The first category of one-shot NAS methods optimizes the architecture and weights in a coupled manner [13,17,154,132,155]. For instance, ENAS [13] uses an LSTM network to discretely sample a new architecture, and then uses a few batches of the training data to optimize the weight of

this architecture. After repeating the above steps many times, a collection of architectures and their corresponding performances are recorded. Finally, the best-performing architecture is selected for further retraining. DARTS [17] uses a similar weight sharing strategy, but has a continuously parameterized architecture distribution. The supernet contains all candidate operations, each with learnable parameters. The best architecture can be directly derived from the distribution. However, as DARTS [17] directly optimizes the supernet weights and the architecture distribution, it suffers from vast GPU memory consumption. Although DARTS-like methods [132,154,155] have adopted different approaches to reduce the resource requirements, coupled optimization inevitably introduces a bias in both architecture distribution and supernet weights [197,229], as they treat all subnetworks unequally. The rapidly converged architectures can easily obtain more opportunities to be optimized [17,159], and are only a small portion of all candidates; therefore, it is challenging to find the best architecture.

Another disadvantage of coupling optimization is that when new architectures are sampled and trained continuously, the weights of previous architectures are negatively impacted, leading to performance degradation. The authors in [230] defined this phenomenon as *multimodel forgetting*. To overcome this problem, Zhang et al. [213] modeled supernet training as a constrained optimization problem of continual learning and proposed *novel search-based architecture selection* (NSAS) loss function. They applied the proposed method to RandomNAS [180] and GDAS [154], where the experimental result demonstrated that the method effectively reduces the multimodel forgetting and boosting the predictive ability of the supernet as an evaluator.

Decoupled optimization. The second category of one-shot NAS methods [209,231,229,217] decouples the optimization of architecture and weights into two sequential phases: (1) training the supernet and (2) using the trained supernet as a predictive performance estimator of different architectures to select the most promising architecture.

In terms of the supernet training phase, the supernet cannot be directly trained as a regular neural network because its weights are also deeply coupled [197]. Yu et al. [11] experimentally showed that the weight-sharing strategy degrades the individual architecture's performance and negatively impacts the real performance ranking of the candidate architectures. To reduce the weight coupling, many one-shot NAS methods [197,209,14,214] adopt the random sampling policy, which randomly samples an architecture from the supernet, activating and optimizing only the weights of this architecture. Meanwhile, RandomNAS [180] demonstrates that a random search policy is a competitive baseline method. Although some one-shot approaches [154,13,155,132,131] have adopted the strategy that samples and trains only one path of the supernet at a time, they sample the path according to the RL controller [13], Gumbel Softmax [154,155,131], or the BinaryConnect network [132], which instead highly couples the architecture and supernet weights. SMASH [14] adopts an auxiliary hypernetwork to generate weights for randomly sampled architectures. Similarly, Zhang et al. [214] proposed a computation graph representation, and used the graph hypernetwork (GHN) to predict the weights for all possible architectures faster and more accurately than regular hypernetworks [14]. However, through a careful experimental analysis conducted to understand the weight-sharing strategy's mechanism, Bender et al. [231] showed that neither a hypernetwork nor an RL controller is required to find the optimal architecture. They proposed a *path dropout* strategy to alleviate the problem of weight coupling. During supernet training, each path of the supernet is randomly dropped with gradually increasing probability. GreedyNAS [216] adopts a multipath sampling strategy to train the greedy supernet. This strategy focuses on more potentially suitable paths, and is demonstrated to effectively achieve a fairly high rank correlation of candidate architectures compared with RS.

The second phase involves the selection of the most promising architecture from the trained supernet, which is the primary purpose of most NAS tasks. Both SMASH [14] and [231] randomly selected a set of architectures from the supernet, and ranked them according to their performance. SMASH can obtain the validation performance of all selected architectures at the cost of a single training run for each architecture, as these architectures are assigned the weights generated by the hypernetwork. Besides, the authors in [231] observed that the architectures with a smaller symmetrized KL divergence value are more likely to perform better. This can be expressed as follows:

$$\begin{aligned} &D_{\mathrm{SKL}} = D_{\mathrm{KL}}(p \parallel q) + D_{\mathrm{KL}}(q \parallel p) \\ &s.t.\ D_{\mathrm{KL}}(p \parallel q) = \sum_{i=1}^{n} p_i \log \frac{p_i}{q_i} \end{aligned} \tag{11}$$

where $(p_1, \ldots, p_n)$ and $(q_1, \ldots, q_n)$ indicate the predictions of the sampled architecture and one-shot model, respectively, and n indicates the number of classes. The cost of calculating the KL value is very small; in [231], only 64 random training data examples were used. Meanwhile, EA is also a promising search solution [197,216]. For instance, SPOS [197] uses EA to search for architectures from the supernet. It is more efficient than the EA methods introduced in Section 4, because each sampled architecture only performs inference. The self-evaluated template network (SETN) [209] proposes an *estimator* to predict the probability of each architecture having a lower validation loss. The experimental results show that SETN can potentially find an architecture with better performance than RS-based methods [231,14].

6.4. Joint hyperparameter and architecture optimization

Most NAS methods fix the same setting of training-related hyperparameters during the whole search stage. After the search, the hyperparameters of the best-performing architecture are further optimized. However, this paradigm may result in suboptimal results as different architectures tend to fit different hyperparameters, making the model ranking unfair [232]. Therefore, a promising solution is the joint *hyperparameter and architecture optimization (HAO)* [34,233,232,234]. We summary the existing joint HAO methods as follows.

Zela et al. [34] cast NAS as a hyperparameter optimization problem, where the search spaces of NAS and standard hyperparameters are combined. They applied BOHB [38], an efficient HPO method, to optimize the architecture and hyperparameters jointly. Similarly, Dong et al. [232] proposed a differentiable method, namely AutoHAS, which builds a Cartesian product of the search spaces of both NAS and HPO by unifying the representation of all candidate choices for the architecture (e.g., number of layers) and hyperparameters (e.g., learning rate). However, a challenge here is that the candidate choices for the architecture search space are usually categorical, while hyperparameters choices can be categorical (e.g., the type of optimizer) and continuous (e.g., learning rate). To overcome this challenge, AutoHAS discretizes the continuous hyperparameters into a linear combination of multiple categorical bases. For example, the categorical bases for the learning rate are $\{0.1, 0.2, 0.3\}$, and then, the final learning rate is defined as $lr = w_1 \times 0.1 + w_2 \times 0.2 + w_3 \times 0.3$. Meanwhile, FBNetv3 [234] jointly searches both architectures and the corresponding training recipes (i.e., hyperparameters). The architectures are represented with one-hot categorical variables and integral (min–max normalized) range variables, and the representation is fed to an encoder network to generate the

architecture embedding. Then, the concatenation of architecture embedding and the training hyperparameters is used to train the accuracy predictor, which will be applied to search for promising architectures and hyperparameters at a later stage.

6.5. Resource-aware NAS

Early NAS studies [12,15,26] pay more attention to searching for neural architectures that achieve higher performance (e.g., classification accuracy), regardless of the associated resource consumption (i.e., the number of GPUs and time required). Therefore, many follow-up studies investigate resource-aware algorithms to trade off performance against the resource budget. To do so, these algorithms add computational cost to the loss function as a resource constraint. These algorithms differ in the type of computational cost, which may be (1) the parameter size; (2) the number of Multiply-ACcumulate (MAC) operations; (3) the number of float-point operations (FLOPs); or (4) the real latency. For example, MONAS [235] considers MAC as the constraint, and as MONAS uses a policy-based reinforcement-learning algorithm to search, the constraint can be directly added to the reward function. MnasNet [130] proposes a customized weighted product to approximate a Pareto optimal solution:

$$\underset{m}{\text{maximize}} \quad ACC(m) \times \left[\frac{LAT(m)}{T}\right]^{w} \tag{12}$$

where $LAT(m)$ denotes measured inference latency of the model m on the target device, T is the target latency, and w is the weight variable defined as:

$$w = \begin{cases} \alpha, & \text{if } LAT(m) \le T \\ \beta, & \text{otherwise} \end{cases} \tag{13}$$

where the recommended value for both α and β is -0.07.

In terms of a differentiable neural architecture search (DNAS) framework, the constraint (i.e., loss function) should be differentiable. For this purpose, FBNet [131] uses a latency lookup table model to estimate the overall latency of a network based on the runtime of each operator. The loss function is defined as

$$\mathcal{L}(a, \theta_a) = \text{CE}(a, \theta_a) \cdot \alpha \log(\text{LAT}(a))^{\beta} \tag{14}$$

where $CE(a, \theta_a)$ indicates the cross-entropy loss of architecture a with weights θ_a. Similar to MnasNet [130], this loss function also comprises two hyperparameters that need to be set manually: α and β control the magnitude of the loss function and the latency term, respectively. In SNAS [155], the cost of time for the generated child network is linear to the one-hot random variables, such that the resource constraint's differentiability is ensured.

7. Open problems and future directions

This section discusses several open problems of the existing AutoML methods and proposes some future research directions.

7.1. Flexible search space

As summarized in Section 4, there are various search spaces where the primitive operations can be roughly classified into pooling and convolution. Some spaces even use a more complex module (e.g., MBConv [130]) as the primitive operation. Although these search spaces have been proven effective for generating well-performing neural architectures, all of them are based on human knowledge and experience, which inevitably introduce human bias, and hence, still do not break away from the human design paradigm. AutoML-Zero [287] uses very simple mathematical operations (e.g., cos, sin, mean,std) as the primitive operations of the search space to minimize the human bias, and applies EA to discover complete machine learning algorithms. AutoML-Zero successfully designs two-layer neural networks based on these basic mathematical operations. Although the network searched by AutoML-Zero is much simpler than both human-designed and NAS-designed networks, the experimental results show the potential to discover a new model design paradigm with minimal human design. Therefore, the design of a more general, flexible, and free of human bias search space and the discovery of novel neural architectures based on this search space would be challenging and advantageous.

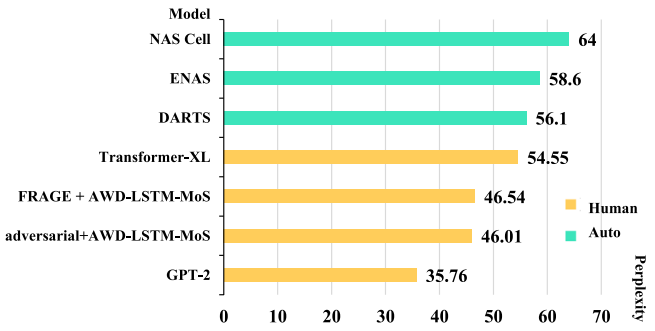


Fig. 20. State-of-the-art models on the PTB dataset. The lower the perplexity, the better is the performance. The green bar represents the automatically generated model, and the yellow bar represents the model designed by human experts. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

7.2. Exploring more areas

As described in Section 6, the models designed by NAS algorithms have achieved comparable results in image classification tasks (CIFAR-10 and ImageNet) to those of manually designed models. Additionally, many recent studies have applied NAS to other CV tasks (Table 5).

However, in terms of the NLP task, most NAS studies have only conducted experiments on the PTB dataset. Besides, some NAS studies have attempted to apply NAS to other NLP tasks (shown in Table 5). However, Fig. 20 shows that, even on the PTB dataset, there is still a big gap in performance between the NAS-designed models [13,17,12] and human-designed models (GPT-2 [288], FRAGE AWD-LSTM-Mos [4], adversarial AWD-LSTM-Mos [289] and Transformer-XL [5]). Therefore, the NAS community still has a long way to achieve comparable results to those of the models designed by experts on NLP tasks.

Besides the CV and NLP tasks, Table 5 also shows that AutoML technique has been applied to other tasks, such as network compression, federate learning, image caption, recommendation system, and searching for loss and activation functions. Therefore, these interesting studies have indicated the potential of AutoML to be applied in more areas.

7.3. Interpretability

Although AutoML algorithms can find promising configuration settings more efficiently than humans, there is a lack of scientific evidence for illustrating why the found settings perform better. For example, in BlockQNN [16], it is unclear why the NAS algorithm tends to select the concatenation operation to process the output of each block in the cell, instead of the element-wise addition operation. Some recent studies [231,290,96] have shown that the explanation for these occurrences is usually hindsight

Table 5
Summary of the existing automated machine learning applications.

Category	Application	References
Computer Vision (CV)	Medical Image Recognition	[236,237]
	Object Detection	[238–243]
	Semantic Segmentation	[244,129,245–249]
	Person Re-identification	[250]
	Super-Resolution	[251–253]
	Image Restoration	[254]
	Generative Adversarial Network (GAN)	[255–258]
	Disparity Estimation	[259]
	Video Task	[260–263]
Natural Language Processing (NLP)	Translation	[264]
	Language Modeling	[265]
	Entity Recognition	[265]
	Text Classification	[266]
	Sequential Labeling	[266]
	Keyword Spotting	[267]
Others	Network Compression	[268–275]
	Graph Neural Network (GNN)	[276]
	Federate Learning	[277,278]
	Loss Function Search	[279,280]
	Activation Function Search	[281]
	Image Caption	[282,283]
	Text to Speech (TTS)	[202]
	Recommendation System	[284–286]

and lacks rigorous mathematical proof. Therefore, increasing the mathematical interpretability of AutoML is an important future research direction.

7.4. Reproducibility

A major challenge with ML is reproducibility. AutoML is no exception, especially for NAS, because most of the existing NAS algorithms still have many parameters that need to be set manually at the implementation level; however, the original papers do not cover much detail. For instance, Yang et al. [123] experimentally demonstrated that the seed plays an important role in NAS experiments; however, most NAS studies do not mention the seed set in the experiments. Besides, considerable resource consumption is another obstacle to reproduction. In this context, several NAS-Bench datasets have been proposed, such as NAS-Bench-101 [224], NAS-Bench-201 [225], and NAS-Bench-NLP [226]. These datasets allow NAS researchers to focus on the design of optimization algorithms without wasting much time on the model evaluation.

7.5. Robustness

NAS has been proven effective in searching promising architectures on many open datasets (e.g., CIFAR-10 and ImageNet). These datasets are generally used for research; therefore, most of the images are well-labeled. However, in real-world situations, the data inevitably contain noise (e.g., mislabeling and inadequate information). Even worse, the data might be modified to be adversarial with carefully designed noises. Deep learning models can be easily fooled by adversarial data, and so can NAS.

So far, there are a few studies [291–294] have attempted to boost the robustness of NAS against adversarial data. Guo et al. [292] experimentally explored the intrinsic impact of network architectures on network robustness against adversarial attacks, and observed that densely connected architectures tend to be more robust. They also found that the flow of solution procedure (FSP) matrix [295] is a good indicator of network robustness, i.e., the lower is the FSP matrix loss, the more robust is the network. Chen et al. [293] proposed a robust loss function for effectively alleviating the performance degradation under symmetric label noise. The authors in [294] adopted EA to search for robust architectures from a well-designed and vast search space, where various adversarial attacks are used as the fitness function for evaluating the robustness of neural architectures.

7.6. Joint hyperparameter and architecture optimization

Most NAS studies have considered HPO and AO as two separate processes. However, as already noted in Section 4, there is a tremendous overlap between the methods used in HPO and AO, e.g., both of them apply RS, BO, and GO methods. In other words, it is feasible to jointly optimize both hyperparameters and architectures, which is experimentally confirmed by several studies [233,232,234]. Thus, how to solve the problem of joint hyperparameter and architecture optimization (HAO) elegantly is a worthy studying issue.

7.7. Complete AutoML pipeline

So far, many AutoML pipeline libraries have been proposed, but most of them only focus on some parts of the AutoML pipeline (Fig. 1). For instance, TPOT [296], Auto-WEAK [177], and Auto-Sklearn [297] are built on top of scikit-learn [298] for building classification and regression pipelines, but they only search for the traditional ML models (such as SVM and KNN). Although TPOT involves neural networks (using Pytorch [299] backend), it only supports an MLP network. Besides, Auto-Keras [22] is an open-source library developed based on Keras [300], which focuses more on searching for deep learning models and supports multi-modal and multi-task. NNI [301] is a more powerful and lightweight toolkit of AutoML, as its built-in capability contains automated feature engineering, hyperparameter optimization, and neural architecture search. Additionally, the NAS module in NNI supports both Pytorch [299] and Tensorflow [302] and reproduces many SOTA NAS methods [13,17,132,128,197,180,224], which is very friendly for NAS researchers and developers. Besides, NNI also integrates scikit-learn features [298], which is one step closer to achieving a complete pipeline. Similarly, Vega [303] is another AutoML algorithm tool that constructs a complete pipeline covering a set of highly decoupled functions: data augmentation, HPO, NAS, model compression, and full training. In summary, designing an easy-to-use and complete AutoML pipeline system is a promising research direction.

7.8. Lifelong learning

Finally, most AutoML algorithms focus only on solving a specific task on some fixed datasets, e.g., image classification on CIFAR-10 and ImageNet. However, a high-quality AutoML system should have the capability of lifelong learning, i.e., it should be able to (1) efficiently **learn new data** and (2) **remember old knowledge**.

7.8.1. Learn new data

First, the system should be able to reuse prior knowledge to solve new tasks (i.e., learning to learn). For example, a child can quickly identify tigers, rabbits, and elephants after seeing several pictures of these animals. However, the current DL models must be trained on considerable data before they can correctly identify images. A hot topic in this area is meta-learning, which aims to design models for new tasks using previous experience.

Meta-learning. Most of the existing NAS methods can search a well-performing architecture for a single task. However, they have to search for a new architecture on a new task; otherwise, the old architecture might not be optimal. Several studies [304–307] have combined meta-learning and NAS to solve this problem. Recently, Lian et al. [306] proposed a novel and meta-learning-based *transferable neural architecture search* method to generate a meta-architecture, which can adapt to new tasks easily and quickly through a few gradient steps. Another challenge of learning new data is few-shot learning scenarios, where there are only limited data for the new tasks. To overcome this challenge, the authors in [305] and [304] applied NAS to few-shot learning, where they only searched for the most promising architecture and optimized it to work on multiple few-shot learning tasks. Elsken et al. [307] proposed a gradient-based meta-learning NAS method, namely METANAS, which can generate *task-specific* architectures more efficiently as it does not require meta-retraining.

Unsupervised learning. Meta-learning-based NAS methods focus more on labeled data, while in some cases, only a portion of the data may have labels or even none at all. Liu et al. [308] proposed a general problem setup, namely *unsupervised neural architecture search (UnNAS)*, to explore whether labels are necessary for NAS. They experimentally demonstrated that the architectures searched without labels are competitive with those searched with labels; therefore, labels are not necessary for NAS, which has provoked some reflection among researchers about which factors do affect NAS.

7.8.2. Remember old knowledge

An AutoML system must be able to constantly learn from new data, without forgetting the knowledge from old data. However, when we use new datasets to train a pretrained model, the model's performance on the previous datasets is substantially reduced. Incremental learning can alleviate this problem. For example, Li and Hoiem [309] proposed the *learning without forgetting* (LwF) method, which trains a model using only new data while preserving its original capabilities. In addition, iCaRL [310] makes progress based on LwF. It only uses a small proportion of old data for pretraining, and then gradually increases the proportion of a new class of data used to train the model.

8. Conclusions

This paper provides a detailed and systematic review of AutoML studies according to the DL pipeline (Fig. 1), ranging from data preparation to model evaluation. Additionally, we compare the performance and efficiency of existing NAS algorithms on the CIFAR-10 and ImageNet datasets, and provide an in-depth discussion of different research directions on NAS: one/two-stage NAS, one-shot NAS, and joint HAO. We also describe several interesting open problems and discuss some important future research directions. Although research on AutoML is in its infancy, we believe that future researchers will effectively solve these problems. In this context, this review provides a comprehensive and clear understanding of AutoML for the benefit of those new to this area, and will thus assist with their future research endeavors.

CRediT authorship contribution statement

Xin He: Writing, Investigation, Data curation, Methodology, Visualization. **Kaiyong Zhao:** Investigation, Writing - review & editing. **Xiaowen Chu:** Conceptualization, Supervision, Writing - review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

The research was supported by the grant RMGS2019_1_23 from Hong Kong Research Matching Grant Scheme. We would like to thank the anonymous reviewers for their valuable comments.

References

[1] A. Krizhevsky, I. Sutskever, G.E. Hinton, ImageNet classification with deep convolutional neural networks, in: International Conference on Neural Information Processing Systems, 2012, pp. 1097–1105.
[2] K. He, X. Zhang, S. Ren, J. Sun, Deep residual learning for image recognition, in: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2016, pp. 770–778.
[3] J. Redmon, S. Divvala, R. Girshick, A. Farhadi, You only look once: Unified, real-time object detection, in: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2016, pp. 779–788.
[4] C. Gong, D. He, X. Tan, T. Qin, L. Wang, T.-Y. Liu, FRAGE: Frequency-agnostic word representation, in: Advances in Neural Information Processing Systems, 2018, pp. 1334–1345.
[5] Z. Dai, Z. Yang, Y. Yang, W.W. Cohen, J. Carbonell, Q.V. Le, R. Salakhutdinov, Transformer-XL: Attentive language models beyond a fixed-length context, 2019, arXiv preprint arXiv:1901.02860.
[6] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A.C. Berg, L. Fei-Fei, ImageNet large scale visual recognition challenge, Int. J. Comput. Vis. 115 (3) (2015) 211–252, http://dx.doi.org/10.1007/s11263-015-0816-y.
[7] K. Simonyan, A. Zisserman, Very deep convolutional networks for large-scale image recognition, 2014, arXiv preprint arXiv:1409.1556.
[8] M. Zoller, M.F. Huber, Benchmark and survey of automated machine learning frameworks, 2019, arXiv preprint arXiv:1904.12054.
[9] Q. Yao, M. Wang, Y. Chen, W. Dai, H. Yi-Qi, L. Yu-Feng, T. Wei-Wei, Y. Qiang, Y. Yang, Taking human out of learning applications: A survey on automated machine learning, 2018, arXiv preprint arXiv:1810.13306.
[10] T. Elsken, J.H. Metzen, F. Hutter, Neural architecture search: A survey, 2018, arXiv preprint arXiv:1808.05377.
[11] C. Sciuto, K. Yu, M. Jaggi, C. Musat, M. Salzmann, Evaluating the search phase of neural architecture search, 2019, arXiv preprint arXiv:1902.08142.
[12] B. Zoph, Q.V. Le, Neural architecture search with reinforcement learning, 2016, arXiv preprint arXiv:1611.01578.
[13] H. Pham, M.Y. Guan, B. Zoph, Q.V. Le, J. Dean, Efficient neural architecture search via parameter sharing, 2018, arXiv preprint arXiv:1802.03268.
[14] A. Brock, T. Lim, J.M. Ritchie, N. Weston, SMASH: One-shot model architecture search through hypernetworks, 2017, arXiv preprint arXiv:1708.05344.
[15] B. Zoph, V. Vasudevan, J. Shlens, Q.V. Le, Learning transferable architectures for scalable image recognition, in: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2018, pp. 8697–8710.
[16] Z. Zhong, J. Yan, W. Wu, J. Shao, C.-L. Liu, Practical block-wise neural network architecture generation, in: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2018, pp. 2423–2432.

[17] H. Liu, K. Simonyan, Y. Yang, Darts: differentiable architecture search, in: International Conference on Learning Representations, 2019, https://openreview.net/forum?id=S1eYHoC5FX.
[18] C. Liu, B. Zoph, M. Neumann, J. Shlens, W. Hua, L.-J. Li, L. Fei-Fei, A. Yuille, J. Huang, K. Murphy, Progressive neural architecture search, in: Proceedings of the European Conference on Computer Vision, ECCV, 2018, pp. 19–34.
[19] H. Liu, K. Simonyan, O. Vinyals, C. Fernando, K. Kavukcuoglu, Hierarchical representations for efficient architecture search, 2017, arXiv preprint arXiv:1711.00436.
[20] T. Chen, I. Goodfellow, J. Shlens, Net2net: Accelerating learning via knowledge transfer, 2015, arXiv preprint arXiv:1511.05641.
[21] T. Wei, C. Wang, Y. Rui, C.W. Chen, Network morphism, in: International Conference on Machine Learning, 2016, pp. 564–572.
[22] H. Jin, Q. Song, X. Hu, Auto-keras: An efficient neural architecture search system, in: Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, ACM, 2019, pp. 1946–1956.
[23] B. Baker, O. Gupta, N. Naik, R. Raskar, Designing neural network architectures using reinforcement learning, 2016, arXiv preprint arXiv:1611.02167.
[24] K.O. Stanley, R. Miikkulainen, Evolving neural networks through augmenting topologies, Evol. Comput. 10 (2) (2002) 99–127.
[25] E. Real, S. Moore, A. Selle, S. Saxena, Y.L. Suematsu, J. Tan, Q.V. Le, A. Kurakin, Large-scale evolution of image classifiers, in: Proceedings of the 34th International Conference on Machine Learning-Volume 70, JMLR. org, 2017, pp. 2902–2911.
[26] E. Real, A. Aggarwal, Y. Huang, Q.V. Le, Regularized evolution for image classifier architecture search, in: Proceedings of the AAAI Conference on Artificial Intelligence, vol. 33, 2019, pp. 4780–4789.
[27] T. Elsken, J.H. Metzen, F. Hutter, Efficient multi-objective neural architecture search via Lamarckian evolution, 2018, arXiv preprint arXiv:1804.09081.
[28] M. Suganuma, S. Shirakawa, T. Nagao, A genetic programming approach to designing convolutional neural network architectures, in: Proceedings of the Genetic and Evolutionary Computation Conference, 2017, pp. 497–504.
[29] R. Miikkulainen, J. Liang, E. Meyerson, A. Rawal, D. Fink, O. Francon, B. Raju, H. Shahrzad, A. Navruzyan, N. Duffy, et al., Evolving deep neural networks, in: Artificial Intelligence in the Age of Neural Networks and Brain Computing, Elsevier, 2019, pp. 293–312.
[30] L. Xie, A. Yuille, Genetic CNN, in: Proceedings of the IEEE International Conference on Computer Vision, 2017, pp. 1379–1388.
[31] K. Ahmed, L. Torresani, Maskconnect: Connectivity learning by gradient descent, in: Proceedings of the European Conference on Computer Vision, ECCV, 2018, pp. 349–365.
[32] R. Shin, C. Packer, D. Song, Differentiable neural network architecture search, in: International Conference on Learning Representations Workshop, 2018.
[33] H. Mendoza, A. Klein, M. Feurer, J.T. Springenberg, F. Hutter, Towards automatically-tuned neural networks, in: Workshop on Automatic Machine Learning, 2016, pp. 58–65.
[34] A. Zela, A. Klein, S. Falkner, F. Hutter, Towards automated deep learning: Efficient joint neural architecture and hyperparameter search, 2018, arXiv preprint arXiv:1807.06906.
[35] A. Klein, S. Falkner, S. Bartels, P. Hennig, F. Hutter, Fast Bayesian optimization of machine learning hyperparameters on large datasets, 2016, arXiv preprint arXiv:1605.07079.
[36] S. Falkner, A. Klein, F. Hutter, Practical Hyperparameter Optimization for Deep Learning, 2018.
[37] F. Hutter, H.H. Hoos, K. Leyton-Brown, Sequential model-based optimization for general algorithm configuration, in: International Conference on Learning and Intelligent Optimization, Springer, 2011, pp. 507–523.
[38] S. Falkner, A. Klein, F. Hutter, BOHB: Robust and efficient hyperparameter optimization at scale, 2018, arXiv preprint arXiv:1807.01774.
[39] J. Bergstra, D. Yamins, D.D. Cox, Making a Science of Model Search: Hyperparameter Optimization in Hundreds of Dimensions for Vision Architectures, JMLR, 2013.
[40] Z. Yang, Y. Wang, X. Chen, B. Shi, C. Xu, C. Xu, Q. Tian, C. Xu, Cars: Continuous evolution for efficient neural architecture search, in: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2020, pp. 1829–1838.
[41] K. Maziarz, M. Tan, A. Khorlin, M. Georgiev, A. Gesmundo, Evolutionary-neural hybrid agents for architecture search, 2018, arXiv:1811.09828.
[42] Y. Chen, G. Meng, Q. Zhang, S. Xiang, C. Huang, L. Mu, X. Wang, Reinforced evolutionary neural architecture search, 2018, arXiv preprint arXiv:1808.00193.
[43] Y. Sun, H. Wang, B. Xue, Y. Jin, G.G. Yen, M. Zhang, Surrogate-assisted evolutionary deep learning using an end-to-end random forest-based performance predictor, IEEE Trans. Evol. Comput. (2019).
[44] B. Wang, Y. Sun, B. Xue, M. Zhang, A hybrid differential evolution approach to designing deep convolutional neural networks for image classification, in: Australasian Joint Conference on Artificial Intelligence, Springer, 2018, pp. 237–250.
[45] M. Wistuba, A. Rawat, T. Pedapati, A survey on neural architecture search, 2019, arXiv preprint arXiv:1905.01392.
[46] P. Ren, Y. Xiao, X. Chang, P.-Y. Huang, Z. Li, X. Chen, X. Wang, A comprehensive survey of neural architecture search: Challenges and solutions, 2020, arXiv:2006.02903.
[47] R. Elshawi, M. Maher, S. Sakr, Automated machine learning: State-of-the-art and open challenges, 2019, arXiv preprint arXiv:1906.02287.
[48] Y. LeCun, L. Bottou, Y. Bengio, P. Haffner, Gradient-based learning applied to document recognition, Proc. IEEE 86 (11) (1998) 2278–2324.
[49] A. Krizhevsky, V. Nair, G. Hinton, The CIFAR-10 dataset, 2014, online: http://www.cs.toronto.edu/kriz/cifar.html.
[50] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, L. Fei-Fei, Imagenet: A large-scale hierarchical image database, in: 2009 IEEE Conference on Computer Vision and Pattern Recognition, IEEE, 2009, pp. 248–255.
[51] J. Yang, X. Sun, Y.-K. Lai, L. Zheng, M.-M. Cheng, Recognition from web data: A progressive filtering approach, IEEE Trans. Image Process. 27 (11) (2018) 5303–5315.
[52] X. Chen, A. Shrivastava, A. Gupta, Neil: Extracting visual knowledge from web data, in: Proceedings of the IEEE International Conference on Computer Vision, 2013, pp. 1409–1416.
[53] Y. Xia, X. Cao, F. Wen, J. Sun, Well begun is half done: Generating high-quality seeds for automatic image dataset construction from web, in: European Conference on Computer Vision, Springer, 2014, pp. 387–400.
[54] N.H. Do, K. Yanai, Automatic construction of action datasets using web videos with density-based cluster analysis and outlier detection, in: Pacific-Rim Symposium on Image and Video Technology, Springer, 2015, pp. 160–172.
[55] J. Krause, B. Sapp, A. Howard, H. Zhou, A. Toshev, T. Duerig, J. Philbin, L. Fei-Fei, The unreasonable effectiveness of noisy data for fine-grained recognition, in: European Conference on Computer Vision, Springer, 2016, pp. 301–320.
[56] P.D. Vo, A. Ginsca, H. Le Borgne, A. Popescu, Harnessing noisy web images for deep representation, Comput. Vis. Image Underst. 164 (2017) 68–81.
[57] B. Collins, J. Deng, K. Li, L. Fei-Fei, Towards scalable dataset construction: An active learning approach, in: European Conference on Computer Vision, Springer, 2008, pp. 86–98.
[58] Y. Roh, G. Heo, S.E. Whang, A survey on data collection for machine learning: A big data-ai integration perspective, IEEE Trans. Knowl. Data Eng. (2019).
[59] D. Yarowsky, Unsupervised word sense disambiguation rivaling supervised methods, in: Proceedings of the 33rd Annual Meeting on Association for Computational Linguistics, Association for Computational Linguistics, 1995, pp. 189–196.
[60] I. Triguero, J.A. Sáez, J. Luengo, S. García, F. Herrera, On the characterization of noise filters for self-training semi-supervised in nearest neighbor classification, Neurocomputing 132 (2014) 30–41.
[61] M.F.A. Hady, F. Schwenker, Combining committee-based semi-supervised learning and active learning, J. Comput. Sci. Tech. 25 (4) (2010) 681–698.
[62] A. Blum, T. Mitchell, Combining labeled and unlabeled data with co-training, in: Proceedings of the Eleventh Annual Conference on Computational Learning Theory, ACM, 1998, pp. 92–100.
[63] Y. Zhou, S. Goldman, Democratic co-learning, in: Tools with Artificial Intelligence, 2004. ICTAI 2004. 16th IEEE International Conference on, IEEE, 2004, pp. 594–602.
[64] X. Chen, A. Gupta, Webly supervised learning of convolutional networks, in: Proceedings of the IEEE International Conference on Computer Vision, 2015, pp. 1431–1439.
[65] Z. Xu, S. Huang, Y. Zhang, D. Tao, Augmenting strong supervision using web data for fine-grained categorization, in: Proceedings of the IEEE International Conference on Computer Vision, 2015, pp. 2524–2532.
[66] N.V. Chawla, K.W. Bowyer, L.O. Hall, W.P. Kegelmeyer, SMOTE: Synthetic minority over-sampling technique, J. Artificial Intelligence Res. 16 (2002) 321–357.
[67] H. Guo, H.L. Viktor, Learning from imbalanced data sets with boosting and data generation: The databoost-im approach, ACM SIGKDD Explor. Newsl. 6 (1) (2004) 30–39.
[68] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, W. Zaremba, Openai gym, 2016, arXiv preprint arXiv:1606.01540.
[69] Q. Wang, S. Zheng, Q. Yan, F. Deng, K. Zhao, X. Chu, IRS: A large synthetic indoor robotics stereo dataset for disparity and surface normal estimation, 2019, arXiv preprint arXiv:1912.09678.
[70] N. Ruiz, S. Schulter, M. Chandraker, Learning to simulate, 2018, arXiv preprint arXiv:1810.02513.
[71] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, Y. Bengio, Generative adversarial nets, in: Advances in Neural Information Processing Systems, 2014, pp. 2672–2680.

[72] T.-H. Oh, R. Jaroensri, C. Kim, M. Elgharib, F. Durand, W.T. Freeman, W. Matusik, Learning-based video motion magnification, in: Proceedings of the European Conference on Computer Vision, ECCV, 2018, pp. 633–648.
[73] L. Sixt, B. Wild, T. Landgraf, Rendergan: generating realistic labeled data, Frontiers in Robotics and AI 5 (2018) 66.
[74] C. Bowles, L. Chen, R. Guerrero, P. Bentley, R. Gunn, A. Hammers, D.A. Dickie, M.V. Hernández, J. Wardlaw, D. Rueckert, Gan augmentation: Augmenting training data using generative adversarial networks, 2018, arXiv preprint arXiv:1810.10863.
[75] N. Park, M. Mohammadi, K. Gorde, S. Jajodia, H. Park, Y. Kim, Data synthesis based on generative adversarial networks, Proc. VLDB Endow. 11 (10) (2018) 1071–1083.
[76] L. Xu, K. Veeramachaneni, Synthesizing tabular data using generative adversarial networks, 2018, arXiv preprint arXiv:1811.11264.
[77] D. Donahue, A. Rumshisky, Adversarial text generation without reinforcement learning, 2018, arXiv preprint arXiv:1810.06640.
[78] T. Karras, S. Laine, T. Aila, A style-based generator architecture for generative adversarial networks, 2018, arXiv preprint arXiv:1812.04948.
[79] X. Chu, I.F. Ilyas, S. Krishnan, J. Wang, Data cleaning: Overview and emerging challenges, in: Proceedings of the 2016 International Conference on Management of Data, 2016, pp. 2201–2206.
[80] M. Jesmeen, J. Hossen, S. Sayeed, C. Ho, K. Tawsif, A. Rahman, E. Arif, A survey on cleaning dirty data using machine learning paradigm for big data analytics, Indones. J. Electr. Eng. Comput. Sci. 10 (3) (2018) 1234–1243.
[81] X. Chu, J. Morcos, I.F. Ilyas, M. Ouzzani, P. Papotti, N. Tang, Y. Ye, Katara: A data cleaning system powered by knowledge bases and crowdsourcing, in: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, ACM, 2015, pp. 1247–1261.
[82] S. Krishnan, J. Wang, M.J. Franklin, K. Goldberg, T. Kraska, T. Milo, E. Wu, SampleClean: Fast and reliable analytics on dirty data, IEEE Data Eng. Bull. 38 (3) (2015) 59–75.
[83] S. Krishnan, M.J. Franklin, K. Goldberg, J. Wang, E. Wu, Activeclean: An interactive data cleaning framework for modern machine learning, in: Proceedings of the 2016 International Conference on Management of Data, 2016, pp. 2117–2120.
[84] S. Krishnan, M.J. Franklin, K. Goldberg, E. Wu, Boostclean: Automated error detection and repair for machine learning, 2017, arXiv preprint arXiv:1711.01299.
[85] S. Krishnan, E. Wu, AlphaClean: Automatic generation of data cleaning pipelines, 2019, arXiv preprint arXiv:1904.11827.
[86] I. Gemp, G. Theocharous, M. Ghavamzadeh, Automated data cleansing through meta-learning, in: Twenty-Ninth IAAI Conference, 2017.
[87] I.F. Ilyas, Effective data cleaning with continuous evaluation, IEEE Data Eng. Bull. 39 (2) (2016) 38–46.
[88] M. Mahdavi, F. Neutatz, L. Visengeriyeva, Z. Abedjan, Towards automated data cleaning workflows, Mach. Learn. 15 (2019) 16.
[89] T. DeVries, G.W. Taylor, Improved regularization of convolutional neural networks with cutout, 2017, arXiv preprint arXiv:1708.04552.
[90] H. Zhang, M. Cisse, Y.N. Dauphin, D. Lopez-Paz, Mixup: Beyond empirical risk minimization, 2017, arXiv preprint arXiv:1710.09412.
[91] A.B. Jung, K. Wada, J. Crall, S. Tanaka, J. Graving, C. Reinders, S. Yadav, J. Banerjee, G. Vecsei, A. Kraft, Z. Rui, J. Borovec, C. Vallentin, S. Zhydenko, K. Pfeiffer, B. Cook, I. Fernández, F.-M. De Rainville, C.-H. Weng, A. Ayala-Acevedo, R. Meudec, M. Laporte, et al., Imgaug, 2020, https://github.com/aleju/imgaug. Online; (accessed 1 February 2020).
[92] A. Buslaev, A. Parinov, E. Khvedchenya, V.I. Iglovikov, A.A. Kalinin, Albumentations: Fast and flexible image augmentations, 2018, ArXiv e-prints, arXiv:1809.06839.
[93] A. Mikołajczyk, M. Grochowski, Data augmentation for improving deep learning in image classification problem, in: 2018 International Interdisciplinary PhD Workshop, IIPhDW, IEEE, 2018, pp. 117–122.
[94] A. Mikołajczyk, M. Grochowski, Style transfer-based image synthesis as an efficient regularization technique in deep learning, in: 2019 24th International Conference on Methods and Models in Automation and Robotics, MMAR, IEEE, 2019, pp. 42–47.
[95] A. Antoniou, A. Storkey, H. Edwards, Data augmentation generative adversarial networks, 2017, arXiv preprint arXiv:1711.04340.
[96] S.C. Wong, A. Gatt, V. Stamatescu, M.D. McDonnell, Understanding data augmentation for classification: When to warp? 2016, arXiv preprint arXiv:1609.08764.
[97] Z. Xie, S.I. Wang, J. Li, D. Lévy, A. Nie, D. Jurafsky, A.Y. Ng, Data noising as smoothing in neural network language models, 2017, arXiv preprint arXiv:1703.02573.
[98] A.W. Yu, D. Dohan, M.-T. Luong, R. Zhao, K. Chen, M. Norouzi, Q.V. Le, QANet: Combining local convolution with global self-attention for reading comprehension, 2018, arXiv preprint arXiv:1804.09541.
[99] E. Ma, NLP augmentation, 2019, https://github.com/makcedward/nlpaug.
[100] E.D. Cubuk, B. Zoph, D. Mane, V. Vasudevan, Q.V. Le, Autoaugment: Learning augmentation strategies from data, in: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2019, pp. 113–123.
[101] Y. Li, G. Hu, Y. Wang, T. Hospedales, N.M. Robertson, Y. Yang, DADA: Differentiable automatic data augmentation, 2020, arXiv preprint arXiv:2003.03780.
[102] R. Hataya, J. Zdenek, K. Yoshizoe, H. Nakayama, Faster AutoAugment: Learning augmentation strategies using backpropagation, 2019, arXiv preprint arXiv:1911.06987.
[103] S. Lim, I. Kim, T. Kim, C. Kim, S. Kim, Fast AutoAugment, in: Advances in Neural Information Processing Systems, 2019, pp. 6662–6672.
[104] A. Naghizadeh, M. Abavisani, D.N. Metaxas, Greedy AutoAugment, 2019, arXiv preprint arXiv:1908.00704.
[105] D. Ho, E. Liang, I. Stoica, P. Abbeel, X. Chen, Population based augmentation: Efficient learning of augmentation policy schedules, 2019, arXiv preprint arXiv:1905.05393.
[106] T. Niu, M. Bansal, Automatically learning data augmentation policies for dialogue tasks, 2019, arXiv preprint arXiv:1909.12868.
[107] M. Geng, K. Xu, B. Ding, H. Wang, L. Zhang, Learning data augmentation policies using augmented random search, 2018, arXiv preprint arXiv:1811.04768.
[108] X. Zhang, Q. Wang, J. Zhang, Z. Zhong, Adversarial AutoAugment, 2019, arXiv preprint arXiv:1912.11188.
[109] C. Lin, M. Guo, C. Li, X. Yuan, W. Wu, J. Yan, D. Lin, W. Ouyang, Online hyper-parameter learning for auto-augmentation strategy, in: Proceedings of the IEEE International Conference on Computer Vision, 2019, pp. 6579–6588.
[110] T.C. LingChen, A. Khonsari, A. Lashkari, M.R. Nazari, J.S. Sambee, M.A. Nascimento, UniformAugment: A search-free probabilistic data augmentation approach, 2020, arXiv preprint arXiv:2003.14348.
[111] H. Motoda, H. Liu, Feature selection, extraction and construction, in: Communication of IICM, vol. 5, (67–72) Institute of Information and Computing Machinery, Taiwan, 2002, p. 2.
[112] M. Dash, H. Liu, Feature selection for classification, Intell. Data Anal. 1 (1–4) (1997) 131–156.
[113] M.J. Pazzani, Constructive induction of cartesian product attributes, in: Feature Extraction, Construction and Selection, Springer, 1998, pp. 341–354.
[114] Z. Zheng, A comparison of constructing different types of new feature for decision tree learning, in: Feature Extraction, Construction and Selection, Springer, 1998, pp. 239–255.
[115] J. Gama, Functional trees, Mach. Learn. 55 (3) (2004) 219–250.
[116] H. Vafaie, K. De Jong, Evolutionary feature space transformation, in: Feature Extraction, Construction and Selection, Springer, 1998, pp. 307–323.
[117] P. Sondhi, Feature Construction Methods: A Survey, 69, sifaka.cs.uiuc.edu, 2009, pp. 70–71.
[118] D. Roth, K. Small, Interactive feature space construction using semantic information, in: Proceedings of the Thirteenth Conference on Computational Natural Language Learning, Association for Computational Linguistics, 2009, pp. 66–74.
[119] Q. Meng, D. Catchpoole, D. Skillicom, P.J. Kennedy, Relational autoencoder for feature extraction, in: 2017 International Joint Conference on Neural Networks, IJCNN, IEEE, 2017, pp. 364–371.
[120] O. Irsoy, E. Alpaydın, Unsupervised feature extraction with autoencoder trees, Neurocomputing 258 (2017) 63–73.
[121] C. Cortes, V. Vapnik, Support-vector networks, Mach. Learn. 20 (3) (1995) 273–297.
[122] N.S. Altman, An introduction to kernel and nearest-neighbor nonparametric regression, Amer. Statist. 46 (3) (1992) 175–185.
[123] A. Yang, P.M. Esperança, F.M. Carlucci, NAS evaluation is frustratingly hard, 2019, arXiv preprint arXiv:1912.12522.
[124] F. Chollet, Xception: Deep learning with depthwise separable convolutions, in: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2017, pp. 1251–1258.
[125] F. Yu, V. Koltun, Multi-scale context aggregation by dilated convolutions, 2015, arXiv preprint arXiv:1511.07122.
[126] J. Hu, L. Shen, G. Sun, Squeeze-and-excitation networks, in: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2018, pp. 7132–7141.
[127] G. Huang, Z. Liu, L. Van Der Maaten, K.Q. Weinberger, Densely connected convolutional networks, in: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2017, pp. 4700–4708.
[128] X. Chen, L. Xie, J. Wu, Q. Tian, Progressive differentiable architecture search: Bridging the depth gap between search and evaluation, in: Proceedings of the IEEE International Conference on Computer Vision, 2019, pp. 1294–1303.

[129] C. Liu, L.-C. Chen, F. Schroff, H. Adam, W. Hua, A.L. Yuille, L. Fei-Fei, Auto-deeplab: Hierarchical neural architecture search for semantic image segmentation, in: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2019, pp. 82–92.
[130] M. Tan, B. Chen, R. Pang, V. Vasudevan, M. Sandler, A. Howard, Q.V. Le, MnasNet: Platform-aware neural architecture search for mobile, in: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2019, pp. 2820–2828.
[131] B. Wu, X. Dai, P. Zhang, Y. Wang, F. Sun, Y. Wu, Y. Tian, P. Vajda, Y. Jia, K. Keutzer, FBNet: Hardware-aware efficient convnet design via differentiable neural architecture search, in: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2019, pp. 10734–10742.
[132] H. Cai, L. Zhu, S. Han, Proxylessnas: Direct neural architecture search on target task and hardware, 2018, arXiv preprint arXiv:1812.00332.
[133] M. Courbariaux, Y. Bengio, J.-P. David, Binaryconnect: Training deep neural networks with binary weights during propagations, in: Advances in Neural Information Processing Systems, 2015, pp. 3123–3131.
[134] G. Hinton, O. Vinyals, J. Dean, Distilling the knowledge in a neural network, 2015, arXiv preprint arXiv:1503.02531.
[135] J. Yosinski, J. Clune, Y. Bengio, H. Lipson, How transferable are features in deep neural networks? in: Advances in Neural Information Processing Systems, 2014, pp. 3320–3328.
[136] T. Wei, C. Wang, C.W. Chen, Modularized morphing of neural networks, 2017, arXiv preprint arXiv:1701.03281.
[137] H. Cai, T. Chen, W. Zhang, Y. Yu, J. Wang, Efficient architecture search by network transformation, in: Thirty-Second AAAI Conference on Artificial Intelligence, 2018.
[138] A. Kwasigroch, M. Grochowski, M. Mikolajczyk, Deep neural network architecture search using network morphism, in: 2019 24th International Conference on Methods and Models in Automation and Robotics, MMAR, IEEE, 2019, pp. 30–35.
[139] H. Cai, J. Yang, W. Zhang, S. Han, Y. Yu, Path-level network transformation for efficient architecture search, 2018, arXiv preprint arXiv:1806.02639.
[140] J. Fang, Y. Sun, K. Peng, Q. Zhang, Y. Li, W. Liu, X. Wang, Fast neural network adaptation via parameter remapping and architecture search, 2020, arXiv preprint arXiv:2001.02525.
[141] A. Gordon, E. Eban, O. Nachum, B. Chen, H. Wu, T.-J. Yang, E. Choi, Morphnet: Fast & simple resource-constrained structure learning of deep networks, in: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2018, pp. 1586–1595.
[142] M. Tan, Q.V. Le, Efficientnet: Rethinking model scaling for convolutional neural networks, 2019, arXiv preprint arXiv:1905.11946.
[143] J.F. Miller, S.L. Harding, Cartesian genetic programming, in: Proceedings of the 10th Annual Conference Companion on Genetic and Evolutionary Computation, ACM, 2008, pp. 2701–2726.
[144] J.F. Miller, S.L. Smith, Redundancy and computational efficiency in cartesian genetic programming, IEEE Trans. Evol. Comput. 10 (2) (2006) 167–174.
[145] F. Gruau, Cellular encoding as a graph grammar, in: IEEE Colloquium on Grammatical Inference: Theory, Applications & Alternatives, 1993.
[146] C. Fernando, D. Banarse, M. Reynolds, F. Besse, D. Pfau, M. Jaderberg, M. Lanctot, D. Wierstra, Convolution by evolution: Differentiable pattern producing networks, in: Proceedings of the Genetic and Evolutionary Computation Conference 2016, ACM, 2016, pp. 109–116.
[147] M. Kim, L. Rigazio, Deep clustered convolutional kernels, in: Feature Extraction: Modern Questions and Challenges, 2015, pp. 160–172.
[148] J.K. Pugh, K.O. Stanley, Evolving multimodal controllers with hyperneat, in: Proceedings of the 15th Annual Conference on Genetic and Evolutionary Computation, ACM, 2013, pp. 735–742.
[149] H. Zhu, Z. An, C. Yang, K. Xu, E. Zhao, Y. Xu, EENA: Efficient evolution of neural architecture, 2019, arXiv:1905.07320.
[150] R.J. Williams, Simple statistical gradient-following algorithms for connectionist reinforcement learning, Mach. Learn. 8 (3–4) (1992) 229–256.
[151] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, O. Klimov, Proximal policy optimization algorithms, 2017, arXiv preprint arXiv:1707.06347.
[152] M. Marcus, G. Kim, M.A. Marcinkiewicz, R. MacIntyre, A. Bies, M. Ferguson, K. Katz, B. Schasberger, The penn treebank: Annotating predicate argument structure, in: Proceedings of the Workshop on Human Language Technology, Association for Computational Linguistics, 1994, pp. 114–119.
[153] C. He, H. Ye, L. Shen, T. Zhang, MiLeNAS: Efficient neural architecture search via mixed-level reformulation, 2020, arXiv:2003.12238.
[154] X. Dong, Y. Yang, Searching for a robust neural architecture in four GPU hours, 2019, arXiv:1910.04465.
[155] S. Xie, H. Zheng, C. Liu, L. Lin, SNAS: Stochastic neural architecture search, 2018, arXiv preprint arXiv:1812.09926.
[156] B. Wu, Y. Wang, P. Zhang, Y. Tian, P. Vajda, K. Keutzer, Mixed precision quantization of convnets via differentiable neural architecture search, 2018, arXiv:1812.00090.
[157] E. Jang, S. Gu, B. Poole, Categorical reparameterization with gumbel-softmax, 2016, arXiv preprint arXiv:1611.01144.
[158] C.J. Maddison, A. Mnih, Y.W. Teh, The concrete distribution: A continuous relaxation of discrete random variables, 2016, arXiv preprint arXiv:1611.00712.
[159] H. Liang, S. Zhang, J. Sun, X. He, W. Huang, K. Zhuang, Z. Li, Darts+: Improved differentiable architecture search with early stopping, 2019, arXiv preprint arXiv:1909.06035.
[160] K. Kandasamy, W. Neiswanger, J. Schneider, B. Poczos, E. Xing, Neural architecture search with Bayesian optimisation and optimal transport, 2018, arXiv:1802.07191.
[161] R. Negrinho, G. Gordon, DeepArchitect: Automatically designing and training deep architectures, 2017, arXiv:1704.08792.
[162] R. Negrinho, D. Patil, N. Le, D. Ferreira, M. Gormley, G. Gordon, Towards modular and programmable architecture search, 2019, arXiv:1909.13404.
[163] G. Dikov, P. van der Smagt, J. Bayer, Bayeslan learning of neural network architectures, 2019, arXiv:1901.04436.
[164] C. White, W. Neiswanger, Y. Savani, BANANAS: Bayesian optimization with neural architectures for neural architecture search, 2019, arXiv:1910.11858.
[165] M. Wistuba, Bayesian optimization combined with incremental evaluation for neural network architecture optimization, in: Proceedings of the International Workshop on Automatic Selection, Configuration and Composition of Machine Learning Algorithms, 2017.
[166] J.-M. Perez-Rua, M. Baccouche, S. Pateux, Efficient progressive neural architecture search, 2018, arXiv:1808.00391.
[167] C.E. Rasmussen, Gaussian processes in machine learning, Lecture Notes in Comput. Sci. (2003) 63–71.
[168] J.S. Bergstra, R. Bardenet, Y. Bengio, B. Kégl, Algorithms for hyper-parameter optimization, in: Advances in Neural Information Processing Systems, 2011, pp. 2546–2554.
[169] R. Luo, F. Tian, T. Qin, E. Chen, T.-Y. Liu, Neural architecture optimization, in: Advances in Neural Information Processing Systems, 2018, pp. 7816–7827.
[170] M.M. Ian Dewancker, S. Clark, Bayesian optimization primer, https://app.sigopt.com/static/pdf/SigOpt_Bayesian_Optimization_Primer.pdf.
[171] B. Shahriari, K. Swersky, Z. Wang, R.P. Adams, N. De Freitas, Taking the human out of the loop: A review of bayesian optimization, Proc. IEEE 104 (1) (2016) 148–175.
[172] J. Snoek, O. Rippel, K. Swersky, R. Kiros, N. Satish, N. Sundaram, M.M.A. Patwary, Prabhat, R.P. Adams, Scalable Bayesian optimization using deep neural networks, 2015, arXiv:1502.05700.
[173] J. Snoek, H. Larochelle, R.P. Adams, Practical Bayesian optimization of machine learning algorithms, in: Advances in Neural Information Processing Systems, 2012, pp. 2951–2959.
[174] J. Stork, M. Zaefferer, T. Bartz-Beielstein, Improving neuroevolution efficiency by surrogate model-based optimization with phenotypic distance kernels, 2019, arXiv:1902.03419.
[175] K. Swersky, D. Duvenaud, J. Snoek, F. Hutter, M.A. Osborne, Raiders of the lost architecture: Kernels for Bayesian optimization in conditional parameter spaces, 2014, arXiv:1409.4011.
[176] A. Camero, H. Wang, E. Alba, T. Bäck, Bayeslan neural architecture search using a training-free performance metric, 2020, arXiv:2001.10726.
[177] C. Thornton, F. Hutter, H.H. Hoos, K. Leyton-Brown, Auto-WEKA: Combined selection and hyperparameter optimization of classification algorithms, in: Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, ACM, 2013, pp. 847–855.
[178] A. sharpdarts, V. Jain, G.D. Hager, sharpdarts: Faster and more Accurate Differentiable Architecture Search, Tech. Rep., 2019.
[179] Y. Geifman, R. El-Yaniv, Deep active learning with a neural architecture search, in: Advances in Neural Information Processing Systems, 2019, pp. 5974–5984.
[180] L. Li, A. Talwalkar, Random search and reproducibility for neural architecture search, 2019, arXiv preprint arXiv:1902.07638.
[181] J. Bergstra, Y. Bengio, Random search for hyper-parameter optimization, J. Mach. Learn. Res. 13 (Feb) (2012) 281–305.
[182] C.-W. Hsu, C.-C. Chang, C.-J. Lin, et al., A Practical Guide To Support Vector Classification, Taipei, 2003.
[183] J.Y. Hesterman, L. Caucci, M.A. Kupinski, H.H. Barrett, L.R. Furenlid, Maximum-likelihood estimation with a contracting-grid search algorithm, IEEE Trans. Nucl. Sci. 57 (3) (2010) 1077–1084.
[184] L. Li, K. Jamieson, G. DeSalvo, A. Rostamizadeh, A. Talwalkar, Hyperband: A novel bandit-based approach to hyperparameter optimization, J. Mach. Learn. Res. 18 (1) (2017) 6765–6816.
[185] M. Feurer, F. Hutter, Hyperparameter optimization, in: Automated Machine Learning: Methods, Systems, Challenges, Springer International Publishing, Cham, 2019, pp. 3–33, http://dx.doi.org/10.1007/978-3-030-05318-5_1.
[186] T. Yu, H. Zhu, Hyper-parameter optimization: A review of algorithms and applications, 2020, arXiv preprint arXiv:2003.05689.
[187] Y. Bengio, Gradient-based optimization of hyperparameters, Neural Comput. 12 (8) (2000) 1889–1900.

[188] J. Domke, Generic methods for optimization-based modeling, in: Artificial Intelligence and Statistics, 2012, pp. 318–326.
[189] D. Maclaurin, D. Duvenaud, R. Adams, Gradient-based hyperparameter optimization through reversible learning, in: International Conference on Machine Learning, 2015, pp. 2113–2122.
[190] F. Pedregosa, Hyperparameter optimization with approximate gradient, 2016, arXiv preprint arXiv:1602.02355.
[191] L. Franceschi, M. Donini, P. Frasconi, M. Pontil, Forward and reverse gradient-based hyperparameter optimization, 2017, arXiv preprint arXiv:1703.01785.
[192] K. Chandra, E. Meijer, S. Andow, E. Arroyo-Fang, I. Dea, J. George, M. Grueter, B. Hosmer, S. Stumpos, A. Tempest, et al., Gradient descent: The ultimate optimizer, 2019, arXiv preprint arXiv:1909.13371.
[193] D.P. Kingma, J. Ba, Adam: A method for stochastic optimization, 2014, arXiv preprint arXiv:1412.6980.
[194] P. Chrabaszcz, I. Loshchilov, F. Hutter, A downsampled variant of imagenet as an alternative to the CIFAR datasets, CoRR abs/1707.08819 (2017) arXiv:1707.08819.
[195] Y.-q. Hu, Y. Yu, W.-w. Tu, Q. Yang, Y. Chen, W. Dai, Multi-Fidelity Automatic Hyper-Parameter Tuning Via Transfer Series Expansion, 2019, p. 8.
[196] C. Wong, N. Houlsby, Y. Lu, A. Gesmundo, Transfer learning with neural automl, in: Advances in Neural Information Processing Systems, 2018, pp. 8356–8365.
[197] D. Stamoulis, R. Ding, D. Wang, D. Lymberopoulos, B. Priyantha, J. Liu, D. Marculescu, Single-path nas: Designing hardware-efficient convnets in less than 4 hours, 2019, arXiv preprint arXiv:1904.02877.
[198] K. Eggensperger, F. Hutter, H.H. Hoos, K. Leyton-Brown, Surrogate benchmarks for hyperparameter optimization., in: MetaSel@ ECAI, 2014, pp. 24–31.
[199] C. Wang, Q. Duan, W. Gong, A. Ye, Z. Di, C. Miao, An evaluation of adaptive surrogate modeling based optimization with two benchmark problems, Environ. Model. Softw. 60 (2014) 167–179.
[200] K. Eggensperger, F. Hutter, H. Hoos, K. Leyton-Brown, Efficient benchmarking of hyperparameter optimizers via surrogates, in: Twenty-Ninth AAAI Conference on Artificial Intelligence, 2015.
[201] K.K. Vu, C. D'Ambrosio, Y. Hamadi, L. Liberti, Surrogate-based methods for black-box optimization, Int. Trans. Oper. Res. 24 (3) (2017) 393–424.
[202] R. Luo, X. Tan, R. Wang, T. Qin, E. Chen, T.-Y. Liu, Semi-supervised neural architecture search, 2020, arXiv:2002.10389.
[203] A. Klein, S. Falkner, J.T. Springenberg, F. Hutter, Learning Curve Prediction with Bayesian Neural Networks, in: International Conference on Learning Representation, 2016.
[204] B. Deng, J. Yan, D. Lin, Peephole: Predicting network performance before training, 2017, arXiv preprint arXiv:1712.03351.
[205] T. Domhan, J.T. Springenberg, F. Hutter, Speeding up automatic hyperparameter optimization of deep neural networks by extrapolation of learning curves, in: Twenty-Fourth International Joint Conference on Artificial Intelligence, 2015.
[206] M. Mahsereci, L. Balles, C. Lassner, P. Hennig, Early stopping without a validation set, 2017, arXiv preprint arXiv:1703.09580.
[207] D. Han, J. Kim, J. Kim, Deep pyramidal residual networks, 2016, arXiv:1610.02915.
[208] J. Cui, P. Chen, R. Li, S. Liu, X. Shen, J. Jia, Fast and practical neural architecture search, in: Proceedings of the IEEE International Conference on Computer Vision, 2019, pp. 6509–6518.
[209] X. Dong, Y. Yang, One-shot neural architecture search via self-evaluated template network, in: Proceedings of the IEEE International Conference on Computer Vision, 2019, pp. 3681–3690.
[210] H. Zhou, M. Yang, J. Wang, W. Pan, BayesNAS: A Bayesian approach for neural architecture search, 2019, arXiv:1905.04919.
[211] Y. Xu, L. Xie, X. Zhang, X. Chen, G.-J. Qi, Q. Tian, H. Xiong, PC-DARTS: Partial channel connections for memory-efficient architecture search, 2019, arXiv:1907.05737.
[212] G. Li, G. Qian, I.C. Delgadillo, M. Muller, A. Thabet, B. Ghanem, SGAS: Sequential greedy architecture search, in: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2020, pp. 1620–1630.
[213] M. Zhang, H. Li, S. Pan, X. Chang, S. Su, Overcoming multi-model forgetting in one-shot NAS with diversity maximization, in: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2020, pp. 7809–7818.
[214] C. Zhang, M. Ren, R. Urtasun, Graph hypernetworks for neural architecture search, 2018, arXiv:1810.05749.
[215] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, L.-C. Chen, MobileNetV2: Inverted residuals and linear bottlenecks, 2018, arXiv:1801.04381.
[216] S. You, T. Huang, M. Yang, F. Wang, C. Qian, C. Zhang, GreedyNAS: Towards fast one-shot NAS with greedy supernet, 2020, arXiv:2003.11236.
[217] H. Cai, C. Gan, S. Han, Once for all: Train one network and specialize it for efficient deployment, 2019, arXiv preprint arXiv:1908.09791.
[218] J. Mei, Y. Li, X. Lian, X. Jin, L. Yang, A. Yuille, J. Yang, AtomNAS: Fine-grained end-to-end neural architecture search, 2019, arXiv preprint arXiv:1912.09640.
[219] S. Hu, S. Xie, H. Zheng, C. Liu, J. Shi, X. Liu, D. Lin, DSNAS: Direct neural architecture search without parameter retraining, 2020, arXiv:2002.09128.
[220] J. Fang, Y. Sun, Q. Zhang, Y. Li, W. Liu, X. Wang, Densely connected search space for more flexible neural architecture search, in: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2020, pp. 10628–10637.
[221] A. Wan, X. Dai, P. Zhang, Z. He, Y. Tian, S. Xie, B. Wu, M. Yu, T. Xu, K. Chen, et al., FBNeTV2: Differentiable neural architecture search for spatial and channel dimensions, in: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2020, pp. 12965–12974.
[222] R. Istrate, F. Scheidegger, G. Mariani, D. Nikolopoulos, C. Bekas, A.C.I. Malossi, TAPAS: Train-less accuracy predictor for architecture search, 2018, arXiv:1806.00250.
[223] M. Kendall, A new measure of rank correlation, Biometrika 30 (1/2) (1938) 81–93, http://www.jstor.org/stable/2332226.
[224] C. Ying, A. Klein, E. Real, E. Christiansen, K. Murphy, F. Hutter, NAS-Bench-101: Towards reproducible neural architecture search, 2019, arXiv e-prints.
[225] X. Dong, Y. Yang, NAS-Bench-201: Extending the scope of reproducible neural architecture search, in: International Conference on Learning Representations, 2020, https://openreview.net/forum?id=HJxyZkBKDr.
[226] N. Klyuchnikov, I. Trofimov, E. Artemova, M. Salnikov, M. Fedorov, E. Burnaev, NAS-Bench-NLP: Neural architecture search benchmark for natural language processing, 2020, arXiv:2006.07116.
[227] X. Zhang, Z. Huang, N. Wang, You only search once: Single shot neural architecture search via direct sparse optimization, 2018, arXiv preprint arXiv:1811.01567.
[228] J. Yu, P. Jin, H. Liu, G. Bender, P.-J. Kindermans, M. Tan, T. Huang, X. Song, R. Pang, Q. Le, BigNAS: Scaling up neural architecture search with big single-stage models, 2020, arXiv preprint arXiv:2003.11142.
[229] X. Chu, B. Zhang, R. Xu, J. Li, Fairnas: Rethinking evaluation fairness of weight sharing neural architecture search, 2019, arXiv preprint arXiv:1907.01845.
[230] Y. Benyahia, K. Yu, K. Bennani-Smires, M. Jaggi, A. Davison, M. Salzmann, C. Musat, Overcoming multi-model forgetting, 2019, arXiv:1902.08232.
[231] G. Bender, P.-J. Kindermans, B. Zoph, V. Vasudevan, Q. Le, Understanding and simplifying one-shot architecture search, in: International Conference on Machine Learning, 2018, pp. 550–559.
[232] X. Dong, M. Tan, A.W. Yu, D. Peng, B. Gabrys, Q.V. Le, AutoHAS: Differentiable hyper-parameter and architecture search, 2020, arXiv:2006.03656.
[233] A. Klein, F. Hutter, Tabular benchmarks for joint architecture and hyperparameter optimization, 2019, arXiv preprint arXiv:1905.04970.
[234] X. Dai, A. Wan, P. Zhang, B. Wu, Z. He, Z. Wei, K. Chen, Y. Tian, M. Yu, P. Vajda, et al., FBNetV3: Joint architecture-recipe search using neural acquisition function, 2020, arXiv preprint arXiv:2006.02049.
[235] C.-H. Hsu, S.-H. Chang, J.-H. Liang, H.-P. Chou, C.-H. Liu, S.-C. Chang, J.-Y. Pan, Y.-T. Chen, W. Wei, D.-C. Juan, MONAS: Multi-objective neural architecture search using reinforcement learning, 2018, arXiv preprint arXiv:1806.10332.
[236] X. He, S. Wang, S. Shi, X. Chu, J. Tang, X. Liu, C. Yan, J. Zhang, G. Ding, Benchmarking deep learning models and automated model design for COVID-19 detection with chest CT scans, medRxiv, 2020.
[237] L. Faes, S.K. Wagner, D.J. Fu, X. Liu, E. Korot, J.R. Ledsam, T. Back, R. Chopra, N. Pontikos, C. Kern, et al., Automated deep learning design for medical image classification by health-care professionals with no coding experience: A feasibility study, Lancet Digit. Health 1 (5) (2019) e232–e242.
[238] G. Ghiasi, T.-Y. Lin, Q.V. Le, NAS-FPN: Learning scalable feature pyramid architecture for object detection, in: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2019, pp. 7036–7045.
[239] H. Xu, L. Yao, W. Zhang, X. Liang, Z. Li, Auto-FPN: Automatic network architecture adaptation for object detection beyond classification, in: Proceedings of the IEEE International Conference on Computer Vision, 2019, pp. 6649–6658.
[240] M. Tan, R. Pang, Q.V. Le, Efficientdet: Scalable and efficient object detection, 2019, arXiv preprint arXiv:1911.09070.
[241] Y. Chen, T. Yang, X. Zhang, G. Meng, X. Xiao, J. Sun, Detnas: backbone search for object detection, in: Advances in Neural Information Processing Systems, 2019, pp. 6642–6652.
[242] J. Guo, K. Han, Y. Wang, C. Zhang, Z. Yang, H. Wu, X. Chen, C. Xu, HitDetector: Hierarchical trinity architecture search for object detection, in: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2020, pp. 11405–11414.

[243] C. Jiang, H. Xu, W. Zhang, X. Liang, Z. Li, SP-NAS: Serial-to-parallel backbone search for object detection, in: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2020, pp. 11863–11872.

[244] Y. Weng, T. Zhou, Y. Li, X. Qiu, NAS-unet: Neural architecture search for medical image segmentation, IEEE Access 7 (2019) 44247–44257.

[245] V. Nekrasov, H. Chen, C. Shen, I. Reid, Fast neural architecture search of compact semantic segmentation models via auxiliary cells, in: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2019, pp. 9126–9135.

[246] W. Bae, S. Lee, Y. Lee, B. Park, M. Chung, K.-H. Jung, Resource optimized neural architecture search for 3D medical image segmentation, in: International Conference on Medical Image Computing and Computer-Assisted Intervention, Springer, 2019, pp. 228–236.

[247] D. Yang, H. Roth, Z. Xu, F. Milletari, L. Zhang, D. Xu, Searching learning strategy with reinforcement learning for 3d medical image segmentation, in: International Conference on Medical Image Computing and Computer-Assisted Intervention, Springer, 2019, pp. 3–11.

[248] N. Dong, M. Xu, X. Liang, Y. Jiang, W. Dai, E. Xing, Neural architecture search for adversarial medical image segmentation, in: International Conference on Medical Image Computing and Computer-Assisted Intervention, Springer, 2019, pp. 828–836.

[249] S. Kim, I. Kim, S. Lim, W. Baek, C. Kim, H. Cho, B. Yoon, T. Kim, Scalable neural architecture search for 3D medical image segmentation, in: International Conference on Medical Image Computing and Computer-Assisted Intervention, Springer, 2019, pp. 220–228.

[250] R. Quan, X. Dong, Y. Wu, L. Zhu, Y. Yang, Auto-reid: Searching for a part-aware convnet for person re-identification, in: Proceedings of the IEEE International Conference on Computer Vision, 2019, pp. 3750–3759.

[251] D. Song, C. Xu, X. Jia, Y. Chen, C. Xu, Y. Wang, Efficient residual dense block search for image super-resolution, in: AAAI, 2020, pp. 12007–12014.

[252] X. Chu, B. Zhang, H. Ma, R. Xu, J. Li, Q. Li, Fast, accurate and lightweight super-resolution with neural architecture search, 2019, arXiv preprint arXiv:1901.07261.

[253] Y. Guo, Y. Luo, Z. He, J. Huang, J. Chen, Hierarchical neural architecture search for single image super-resolution, 2020, arXiv preprint arXiv:2003.04619.

[254] H. Zhang, Y. Li, H. Chen, C. Shen, IR-NAS: Neural architecture search for image restoration, 2019, arXiv preprint arXiv:1909.08228.

[255] X. Gong, S. Chang, Y. Jiang, Z. Wang, Autogan: Neural architecture search for generative adversarial networks, in: Proceedings of the IEEE International Conference on Computer Vision, 2019, pp. 3224–3234.

[256] Y. Fu, W. Chen, H. Wang, H. Li, Y. Lin, Z. Wang, AutoGAN-distiller: Searching to compress generative adversarial networks, 2020, arXiv preprint arXiv:2006.08198.

[257] M. Li, J. Lin, Y. Ding, Z. Liu, J.-Y. Zhu, S. Han, Gan compression: Efficient architectures for interactive conditional gans, in: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2020, pp. 5284–5294.

[258] C. Gao, Y. Chen, S. Liu, Z. Tan, S. Yan, Adversarialnas: Adversarial neural architecture search for gans, in: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2020, pp. 5680–5689.

[259] T. Saikia, Y. Marrakchi, A. Zela, F. Hutter, T. Brox, Autodispnet: Improving disparity estimation with automl, in: Proceedings of the IEEE International Conference on Computer Vision, 2019, pp. 1812–1823.

[260] W. Peng, X. Hong, G. Zhao, Video action recognition via neural architecture searching, in: 2019 IEEE International Conference on Image Processing, ICIP, IEEE, 2019, pp. 11–15.

[261] M.S. Ryoo, A. Piergiovanni, M. Tan, A. Angelova, Assemblenet: Searching for multi-stream neural connectivity in video architectures, 2019, arXiv preprint arXiv:1905.13209.

[262] V. Nekrasov, H. Chen, C. Shen, I. Reid, Architecture search of dynamic cells for semantic video segmentation, in: The IEEE Winter Conference on Applications of Computer Vision, 2020, pp. 1970–1979.

[263] A. Piergiovanni, A. Angelova, A. Toshev, M.S. Ryoo, Evolving space-time neural architectures for videos, in: Proceedings of the IEEE International Conference on Computer Vision, 2019, pp. 1793–1802.

[264] Y. Fan, F. Tian, Y. Xia, T. Qin, X.-Y. Li, T.-Y. Liu, Searching better architectures for neural machine translation, IEEE/ACM Trans. Audio Speech Lang. Process. (2020).

[265] Y. Jiang, C. Hu, T. Xiao, C. Zhang, J. Zhu, Improved differentiable architecture search for language modeling and named entity recognition, in: Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing, EMNLP-IJCNLP, 2019, pp. 3576–3581.

[266] J. Chen, K. Chen, X. Chen, X. Qiu, X. Huang, Exploring shared structures and hierarchies for multiple NLP tasks, 2018, arXiv preprint arXiv:1808.07658.

[267] H. Mazzawi, X. Gonzalvo, A. Kracun, P. Sridhar, N. Subrahmanya, I. Lopez-Moreno, H.-J. Park, P. Violette, Improving keyword spotting and language identification via neural architecture search at scale, in: INTERSPEECH, 2019, pp. 1278–1282.

[268] Y. He, J. Lin, Z. Liu, H. Wang, L.-J. Li, S. Han, Amc: Automl for model compression and acceleration on mobile devices, in: Proceedings of the European Conference on Computer Vision, ECCV, 2018, pp. 784–800.

[269] X. Xiao, Z. Wang, S. Rajasekaran, AutoPrune: Automatic network pruning by regularizing auxiliary parameters, in: Advances in Neural Information Processing Systems, 2019, pp. 13681–13691.

[270] R. Zhao, W. Luk, Efficient structured pruning and architecture searching for group convolution, in: Proceedings of the IEEE International Conference on Computer Vision Workshops, 2019.

[271] T. Wang, K. Wang, H. Cai, J. Lin, Z. Liu, H. Wang, Y. Lin, S. Han, APQ: Joint search for network architecture, pruning and quantization policy, in: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2020, pp. 2078–2087.

[272] X. Dong, Y. Yang, Network pruning via transformable architecture search, in: Advances in Neural Information Processing Systems, 2019, pp. 760–771.

[273] Q. Huang, K. Zhou, S. You, U. Neumann, Learning to prune filters in convolutional neural networks, 2018, arXiv:1801.07365.

[274] Y. He, P. Liu, L. Zhu, Y. Yang, Meta filter pruning to accelerate deep convolutional neural networks, 2019, arXiv:1904.03961.

[275] T.-W. Chin, C. Zhang, D. Marculescu, Layer-compensated pruning for resource-constrained convolutional neural networks, 2018, arXiv:1810.00518.

[276] K. Zhou, Q. Song, X. Huang, X. Hu, Auto-GNN: Neural architecture search of graph neural networks, 2019, arXiv preprint arXiv:1909.03184.

[277] C. He, M. Annavaram, S. Avestimehr, FedNAS: Federated deep learning via neural architecture search, 2020, arXiv:2004.08546.

[278] H. Zhu, Y. Jin, Real-time federated evolutionary neural architecture search, 2020, arXiv preprint arXiv:2003.02793.

[279] C. Li, X. Yuan, C. Lin, M. Guo, W. Wu, J. Yan, W. Ouyang, AM-LFS: Automl for loss function search, in: Proceedings of the IEEE International Conference on Computer Vision, 2019, pp. 8410–8419.

[280] B. Ru, C. Lyle, L. Schut, M. van der Wilk, Y. Gal, Revisiting the train loss: An efficient performance estimator for neural architecture search, 2020, arXiv preprint arXiv:2006.04492.

[281] P. Ramachandran, B. Zoph, Q.V. Le, Searching for activation functions, 2017, arXiv:1710.05941.

[282] H. Wang, H. Wang, K. Xu, Evolutionary recurrent neural network for image captioning, Neurocomputing (2020).

[283] L. Wang, Y. Zhao, Y. Jinnai, Y. Tian, R. Fonseca, Neural architecture search using deep neural networks and Monte Carlo tree search, 2018, arXiv preprint arXiv:1805.07440.

[284] P. Zhao, K. Xiao, Y. Zhang, K. Bian, W. Yan, AMER: Automatic behavior modeling and interaction exploration in recommender system, 2020, arXiv preprint arXiv:2006.05933.

[285] X. Zhao, C. Wang, M. Chen, X. Zheng, X. Liu, J. Tang, AutoEmb: Automated embedding dimensionality search in streaming recommendations, 2020, arXiv preprint arXiv:2002.11252.

[286] W. Cheng, Y. Shen, L. Huang, Differentiable neural input search for recommender systems, 2020, arXiv preprint arXiv:2006.04466.

[287] E. Real, C. Liang, D.R. So, Q.V. Le, AutoML-zero: Evolving machine learning algorithms from scratch, 2020, arXiv:2003.03384.

[288] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever, Language models are unsupervised multitask learners, in: OpenAI Blog, vol. 1, 2019, p. 8.

[289] D. Wang, C. Gong, Q. Liu, Improving neural language modeling via adversarial training, 2019, arXiv preprint arXiv:1906.03805.

[290] A. Zela, T. Elsken, T. Saikia, Y. Marrakchi, T. Brox, F. Hutter, Understanding and robustifying differentiable architecture search, 2019, arXiv:1909.09656.

[291] S. Kotyan, D.V. Vargas, Is neural architecture search a way forward to develop robust neural networks? in: Proceedings of the Annual Conference of JSAI, JSAI2020, 2020, pp. 2K1ES203–2K1ES203.

[292] M. Guo, Y. Yang, R. Xu, Z. Liu, D. Lin, When NAS meets robustness: In search of robust architectures against adversarial attacks, in: IEEE Conference on Computer Vision and Pattern Recognition, CVPR, 2020.

[293] Y. Chen, Q. Song, X. Liu, P.S. Sastry, X. Hu, On robustness of neural architecture search under label noise, in: Frontiers in Big Data, 2020.

[294] D.V. Vargas, S. Kotyan, Evolving robust neural architectures to defend from adversarial attacks, 2019, arXiv preprint arXiv:1906.11667.

[295] J. Yim, D. Joo, J. Bae, J. Kim, A Gift from knowledge distillation: Fast optimization, network minimization and transfer learning, in: 2017 IEEE Conference on Computer Vision and Pattern Recognition, CVPR, 2017, pp. 7130–7138.

[296] G. Squillero, P. Burelli, Applications of evolutionary computation, in: 19th European Conference, EvoApplications 2016, Porto, Portugal, March 30–April 1, 2016, Proceedings, Vol. 9597, Springer, 2016.

[297] M. Feurer, A. Klein, K. Eggensperger, J. Springenberg, M. Blum, F. Hutter, Efficient and robust automated machine learning, in: C. Cortes, N.D. Lawrence, D.D. Lee, M. Sugiyama, R. Garnett (Eds.), Advances in Neural Information Processing Systems 28, Curran Associates, Inc., 2015, pp. 2962–2970.
[298] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, E. Duchesnay, Scikit-learn: Machine learning in Python, J. Mach. Learn. Res. 12 (2011) 2825–2830.
[299] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, S. Chintala, PyTorch: An imperative style, high-performance deep learning library, in: Advances in Neural Information Processing Systems 32, Curran Associates, Inc., 2019, pp. 8024–8035.
[300] F. Chollet, et al., Keras, 2015, https://github.com/keras-team/keras.
[301] NNI (neural network intelligence), 2020, https://github.com/microsoft/nni.
[302] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D.G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, X. Zheng, TensorFlow: A system for large-scale machine learning, 2016, arXiv:1605.08695.
[303] B. Wang, H. Xu, J. Zhang, C. Chen, X. Fang, N. Kang, L. Hong, W. Zhang, Y. Li, Z. Liu, et al., Vega: towards an end-to-end configurable automl pipeline, 2020, arXiv preprint arXiv:2011.01507.
[304] R. Pasunuru, M. Bansal, Continual and multi-task architecture search, 2019, arXiv preprint arXiv:1906.05226.
[305] J. Kim, S. Lee, S. Kim, M. Cha, J.K. Lee, Y. Choi, Y. Choi, D.-Y. Cho, J. Kim, Auto-meta: Automated gradient based meta learner search, 2018, arXiv preprint arXiv:1806.06927.
[306] D. Lian, Y. Zheng, Y. Xu, Y. Lu, L. Lin, P. Zhao, J. Huang, S. Gao, Towards fast adaptation of neural architectures with meta learning, in: International Conference on Learning Representations, 2019.
[307] T. Elsken, B. Staffler, J.H. Metzen, F. Hutter, Meta-Learning of neural architectures for few-shot learning, in: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2020, pp. 12365–12375.
[308] C. Liu, P. Dollár, K. He, R. Girshick, A. Yuille, S. Xie, Are labels necessary for neural architecture search? 2020, arXiv:2003.12056.
[309] Z. Li, D. Hoiem, Learning without forgetting, IEEE Trans. Pattern Anal. Mach. Intell. 40 (12) (2018) 2935–2947.
[310] S.-A. Rebuffi, A. Kolesnikov, G. Sperl, C.H. Lampert, iCaRL: Incremental classifier and representation learning, in: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2017, pp. 2001–2010.