AutoML: A Survey of the State-of-the-Art

Xin He, Kaiyong Zhao, Xiaowen Chu*

Department of Computer Science, Hong Kong Baptist University

Abstract

Deep learning (DL) techniques have penetrated all aspects of our lives and brought us great convenience. However, building a high-quality DL system for a specific task highly relies on human expertise, hindering the applications of DL to more areas. Automated machine learning (AutoML) becomes a promising solution to build a DL system without human assistance, and a growing number of researchers focus on AutoML. In this paper, we provide a comprehensive and up-to-date review of the state-of-the-art (SOTA) in AutoML. First, we introduce AutoML methods according to the pipeline, covering data preparation, feature engineering, hyperparameter optimization, and neural architecture search (NAS). We focus more on NAS, as it is currently very hot sub-topic of AutoML. We summarize the performance of the representative NAS algorithms on the CIFAR-10 and ImageNet datasets and further discuss several worthy studying directions of NAS methods: one/two-stage NAS, one-shot NAS, and joint hyperparameter and architecture optimization. Finally, we discuss some open problems of the existing AutoML methods for future research.

Keywords: deep learning, automated machine learning (AutoML), neural architecture search (NAS), hyperparameter optimization (HPO)

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 Figure 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 Figure 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 estimation 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

*Corresponding author

Email addresses: `csxinhe@comp.hkbu.edu.hk` (Xin He), `kyzhao@comp.hkbu.edu.hk` (Kaiyong Zhao), `chxw@comp.hkbu.edu.hk` (Xiaowen Chu)

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

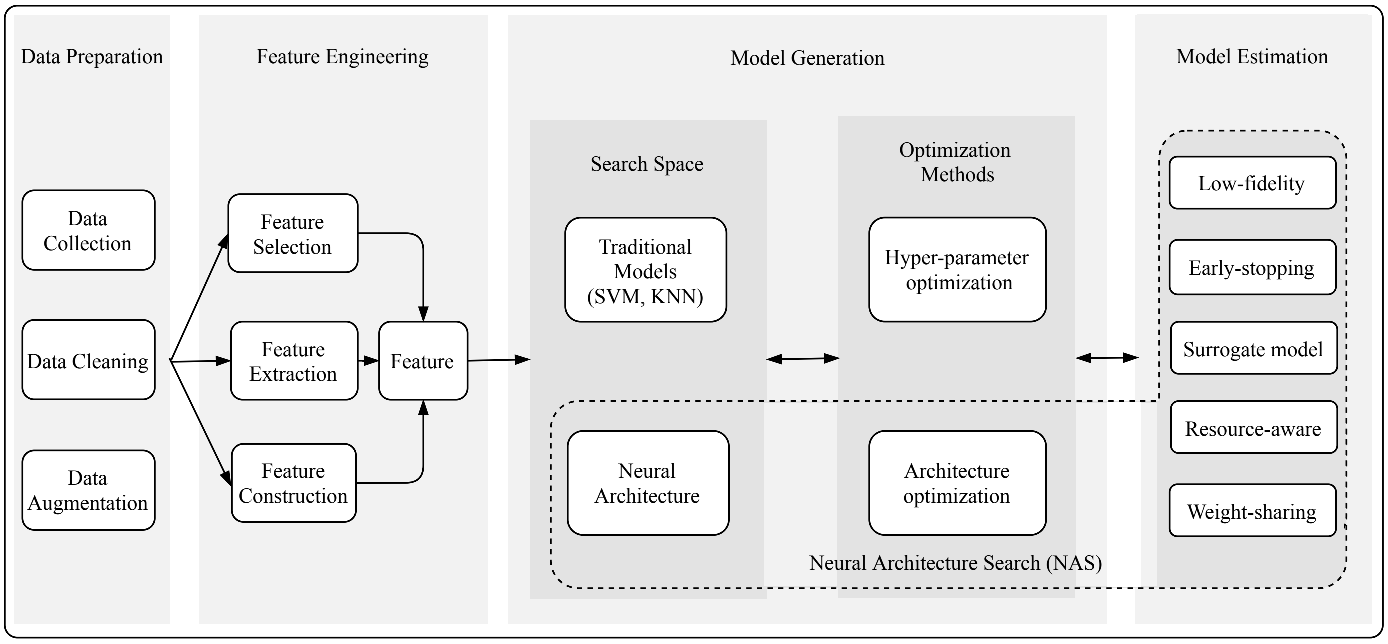


Figure 1: An overview of AutoML pipeline covering data preparation (Section 2), feature engineering (Section 3), model generation (Section 4) and model evaluation (Section 5).

a recurrent network is trained by reinforcement learning to automatically search for the best-performing 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, 13, 14], *cell-based* [13, 15, 16, 17, 18], *hierarchical* [19] and *morphism-based* [20, 21, 22] search space. The commonly used AO methods contain *reinforcement learning* (RL) [12, 15, 23, 16, 13], *evolution-based algorithm* (EA) [24, 25, 26, 27, 28, 29, 30], and *gradient descent* (GD) [17, 31, 32], *Surrogate Model-Based Optimization* (SMBO) [33, 34, 35, 36, 37, 38, 39], and hybrid AO methods [40, 41, 42, 43, 44].

Although there are already several AutoML-related surveys [10, 45, 46, 9, 8], to the best of our knowledge, our survey covers a wider range of AutoML methods. As summarized in Table 1, [10, 45, 46] only focus on NAS, and [9, 8] do not detail NAS technique. In this paper, we propose the AutoML pipeline and summarize the AutoML-related methods according to the pipeline (Figure 1), providing beginners with a comprehensive introduction to AutoML. Notably, many sub-topics of AutoML are large enough to have their 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, 3, 4, 5, respectively. In Section 6, we compare the performance of NAS algorithms on the CIFAR-10 and ImageNet dataset, and discuss two subtopics (one/two-stage and one-shot NAS) of great concern in NAS community. In Section 7, we describe several open problems in AutoML. We conclude our survey in Section 8.

Survey	**DP**	**FE**	**HPO**	**NAS**
[10]	-	-	-	✓
[45]	-	-	-	✓
[46]	-	-	-	✓
[9]	-	✓	✓	†
[47]	✓	-	✓	†
[8]	✓	✓	✓	-
Ours	✓	✓	✓	✓

Table 1: Comparison between different AutoML surveys. *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.

2. Data Preparation

The first step in the ML pipeline is data preparation. Figure 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.

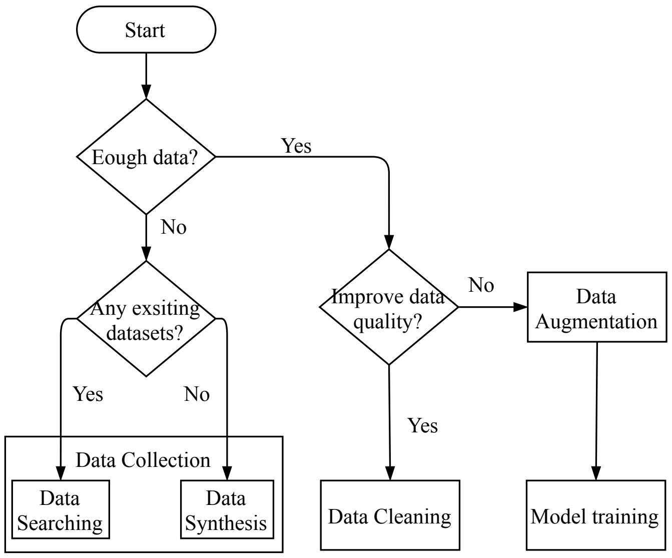


Figure 2: The flow chart for data preparation.

2.1. Data Collection

ML's deepening study has led to a consensus that good data must be available; as a result, numerous open datasets have emerged. In the early stages of ML's 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 also 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 private matters. Two types of methods are proposed to solve this problem: data searching and data synthesis.

2.1.1. Data Searching

As the Internet is an inexhaustible source of data, searching for Web data is an intuitive way to collect a dataset [51, 52, 53, 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] provide 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] assign 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] propose 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] combine 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] use 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 derive synthetic data is *Generative Adversarial Networks* (GANs) [72], which can be used to generate images [72, 73, 74, 75], tabular [76, 77] and text [78] data. Figure 3 shows some human face images,

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

Figure 3: The examples of human face images generated by GAN [71].

which are generated by GAN in the work of Karras et al. [71]. Oh and Jaroensri et al. [73] build a synthetic dataset, which captures small motion for video-motion magnification. Bowles et al. [75] demonstrate 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. [78] use an autoencoder to encode sentences into a smooth sentence representation to remove the barrier of reinforcement learning. Park et al. [76] apply GAN to synthesize fake tables that are statistically similar to the original table but do not cause information leakage. Similarly, in [77], 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] propose Katara, a knowledge-based and crowd-powered data cleaning system. To improve efficiency, some methods [82, 83] propose 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 hyper-parameter optimization problem, which further

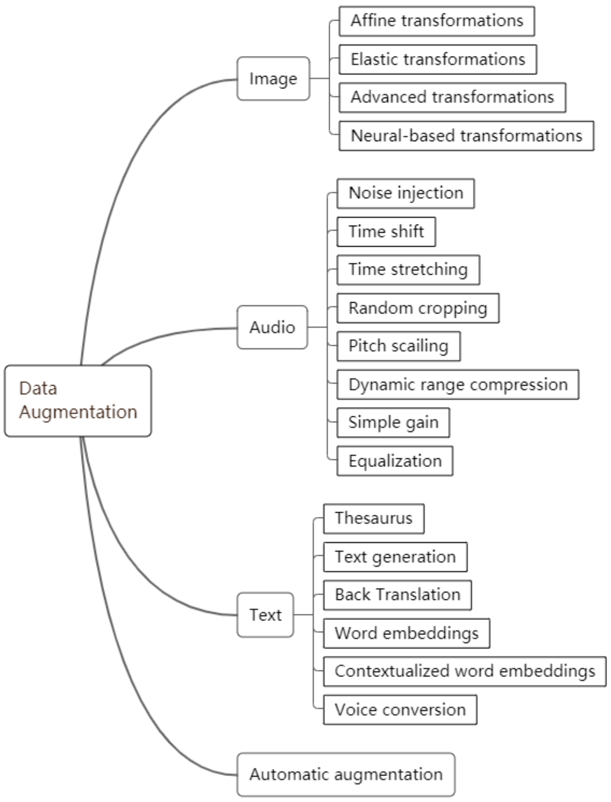


Figure 4: A classification of data augmentation techniques.

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.

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 process becomes a worth studying problem, especially for enterprises. Ilyas et al. [86] propose an effective way of evaluating the algorithms of continuously cleaning data. Mahdavi et al. [87] build a cleaning workflow orchestrator, which can learn from previous cleaning tasks, and propose 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. Figure 4 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 [88], and mixup [89], etc. These three types of common transformations are available in some open source libraries, like torchvision [5], ImageAug [90], and Albumentations [91]. In terms of neural-based transformations, it can be divided into three categories: adversarial noise [92], neural style transfer [93], and GAN technique [94]. For textual data, Wong et al. [95] propose 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. [96] propose 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. [97] propose the use of back-translation for DA to aid reading comprehension. NLPAug [98] 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 [99, 100, 101, 102, 103, 104, 105, 106, 107, 108, 109] proposed to search for augmentation policy for different tasks. AutoAugment [99] 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 [100, 101], Bayesian-based optimization [102], online hyper-parameter learning [108], greedy-based search [103] and random search [106]. Besides, LingChen et al. [109] propose a search-free DA method, namely UniformAugment, by assuming that the augmentation space is approximately distribution invariant.

3. Feature Engineering

In industry, 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 sub-topics: 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 [110]. 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.

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

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 [111], there are four basic steps in a typical process of feature selection (see Figure 5), as follows:

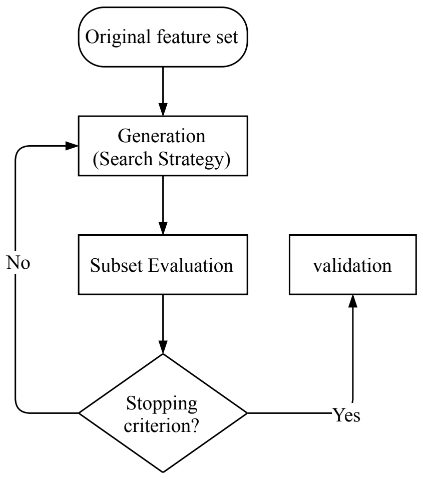


Figure 5: The iterative process of feature selection. A subset of features is selected, based on a search strategy, and then evaluated. Then, a validation procedure is implemented to determine whether the subset is valid. The above steps are repeated until the stop criterion is satisfied.

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 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 [112] and M-of-N [113] are commonly used for nominal features.

It is impossible to manually explore all possibilities. Hence, to further improve efficiency, some automatic feature-construction methods have been proposed and shown to achieve results as good as or superior to those achieved by human expertise. These algorithms are aimed to automate the process of searching and evaluating the operation combination. In terms of searching, algorithms such as decision tree-based methods [114, 113] and genetic algorithms [115] require a pre-defined operation space, while annotation-based approaches do not, as the latter use domain knowledge (in the form of annotation), together with the training examples [116]. Such methods can be traced back to the interactive feature-space construction protocol introduced by [117]. Using this protocol, the learner identifies inadequate regions of feature space and, in coordination with a domain expert, adds descriptiveness using existing semantic resources. Then, after selecting possible operations and constructing a new feature, feature-selection techniques are applied to measure 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 networks 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. [118] propose a relation autoencoder model that considers data features and their relationships, while an unsupervised feature-extraction method using autoencoder trees is proposed by [119].

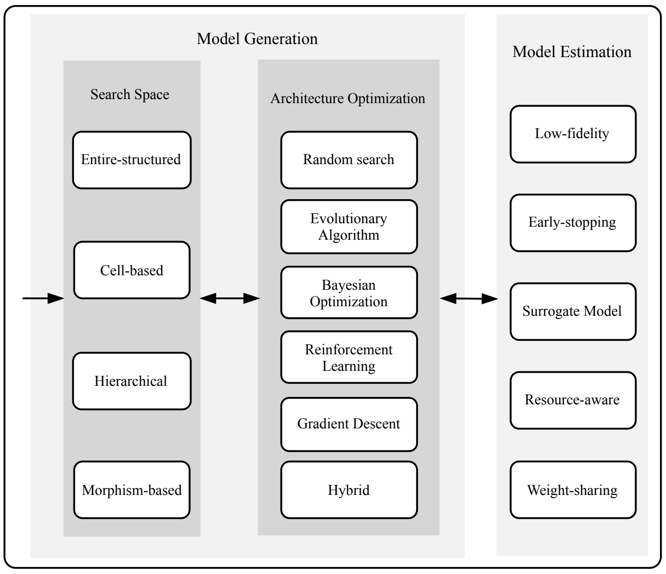


Figure 6: An overview of NAS pipeline.

4. Model Generation

As Figure 1 shows, model generation is divided into two parts: search space and optimization methods. The search space defines the model structures that can be designed and optimized in principle. The types of model can be broadly divided into two categories: one is traditional ML models, such as *support-vector machine (SVM)*, *k-nearest neighbors (KNN)*, and *decision tree*, and the other is deep neural networks (DNNs). In terms of optimization methods, it can also be categorized into two groups: hyper-parameters that are used for training, like the learning rate and batch size, and those used for model design, such as the number of neighbors for KNN or the number of layers for DNN. In this section, we focus more on *Neural Architecture Search (NAS)*, which has got recently much attention. We refer the readers who are interested in traditional model (e.g., SVM) to other reviews [9, 8].

Figure 6 presents an overview of NAS pipeline, which is categorized into the following three dimensions [10, 120]: search space, architecture optimization (AO) method[6], and

[6]It may also be referred to as the "search strategy [10, 120]",

model estimation method. To make the follow-up content easier to understand, we explain three concepts as below.

- *Search Space.* The search space defines the design principles of neural architectures. Different scenarios require different search spaces. We summarize four commonly used search space, including entire-structured, cell-based, hierarchical, and morphism-based search space.
- *Architecture Optimization Method.* The AO method defines how to guide the search to efficiently find the model architecture with high performance after the search space is defined.
- *Model Estimation Method.* Once a model is generated, its performance needs to be evaluated. The simplest way is to train the model to convergence on the training set, and then do the evaluation on the validation set, whereas such method is time-consuming and resource-intensive. Some advanced methods maybe accelerate the process of estimation but loss fidelity. Thus, how to balance efficiency and effectiveness of evaluation is a problem worth studying.

The search space and architecture optimization method are presented in this section, while the methods of model estimation are included in the next section.

4.1. Search Space

The neural architecture can be represented as a direct acyclic graph (DAG) [13, 45], which is formed by ordered nodes Z and by edges connecting pairs of nodes, where each node indicates a tensor z and each edge represent an operation o selected from a set of candidate operations O. The in-degree of each node varies from the design of search space. Due to the limitation of computational resources, a maximum threshold N is manually set for in-degree. Here, we give the formula for the computation at a node k under the assumption that the index for nodes starts from 1:

$$z^{(k)} = \sum_{i=1}^{N} o^{(i)}(z^{(i)}), \ o^{(i)} \in O \tag{1}$$

The set of candidate operations O mainly includes the primitive operation such as convolution, pooling, activation functions, skip connection, concatenation, and addition. Besides, to further enhance the performance of the model, many NAS methods use some advanced human-designed modules as the primitive operations, such as depth-wise separable convolution [121], dilated convolution[122], and Squeeze-and-Excitation (SE) Block [123]. How to select and combine these operations varies from the design search space. In other words, the search space defines the structural paradigm that architecture optimization algorithms can explore, thus designing a good search space is a promising 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 works, we detail the commonly used search spaces as follows.

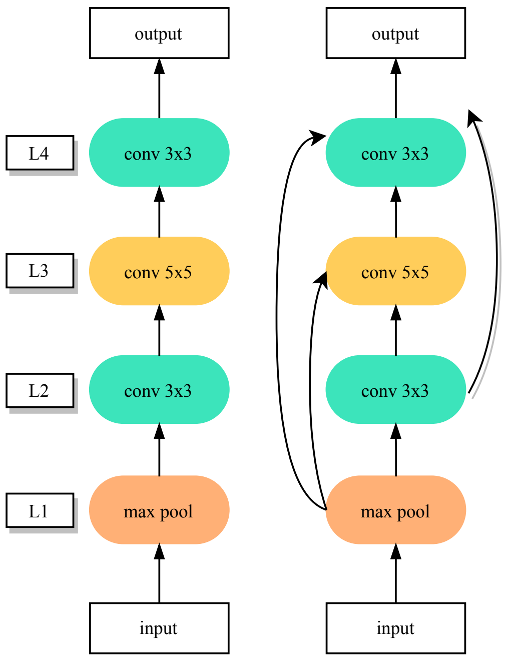


Figure 7: Two simplified examples of the entire-structured neural architectures. Each layer is specified with different operation, such as convolution and max-pooling operations. The edge indicates the flow of information. The skip-connection operation used in the right example can help explore more deeper and 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 space. Figure 7 presents two simplified examples of the entire-structured models, which are built by stacking a predefined number of nodes, where each node represents a layer and has a specified operation. The simplest structure is the left model in Figure 7, while the right model is relatively complex, as it permits arbitrary skip connections [2] to exist between the ordered nodes, and these connections have been proven effective in practice [12]. Although the entire structure is easy to implement, it has several disadvantages. For example, it is widely accepted that the deeper the model, the better the generalization ability, but searching for such a deep network is onerous and computationally expensive. Furthermore, the generated architecture lacks transferability: i.e., 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.

”search policy [11]”, or ”optimization method [45, 9]”.

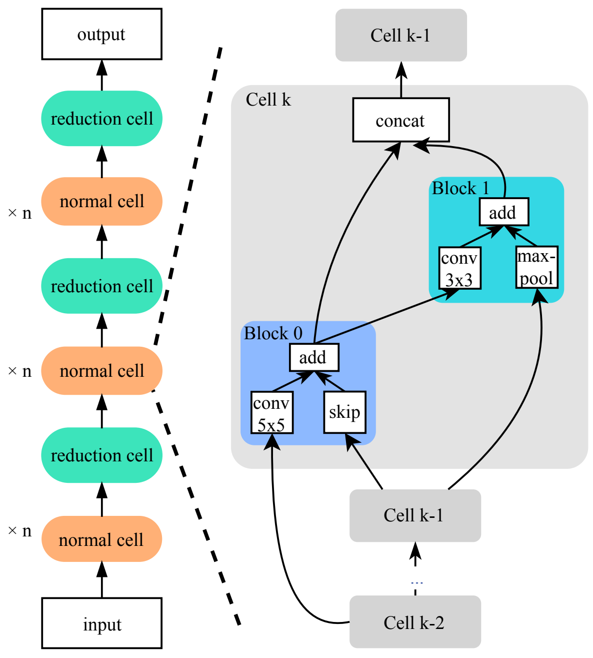


Figure 8: (Left) An example of a cell-based model consisting of three motifs, each with n normal cells and one reduction cell. (Right) An example of normal cell, which contains two blocks, each has two nodes. Each node is specified with different operations and inputs, respectively.

4.1.2. Cell-Based Search Space

Motivation. To enable the transferability of the generated model, many works [15, 16, 13] propose the cell-based search space, 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, 124] are also built by stacking a fixed number of modules. For example, The family of ResNet builds many variants, such as ResNet50, ResNet101, and ResNet152, by stacking more BottleNeck modules [2]. Throughout the literature, this repeated module has been referred to as motif, cell, or block, while in this paper, we call it *cell*.

Design. Figure 8 (left) presents an example of a final cell-based neural network, which consists of two types of cells, namely *normal* and *reduction* cell. Thus, the problem of searching for a full neural architecture is simplified into searching for an optimal cell structure 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 the normal cell, but the difference is that the width and height of output feature maps of the reduction cell is half the input, and the number of channels is twice the number of 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. Many approaches [17, 13, 15] also experimentally demonstrate the transferability of the model generated in cell-based search space, as the model built on the CIFAR-10 can also achieve comparable results to SOTA human-designed models on the ImageNet.

The design paradigm of the internal cell structure of most works refers to Zoph et al. [15], which is one of the first to propose an exploration of cell-based search space. Figure 8 (right) shows an example of the normal cell structure. Each cell contains B blocks (here $B = 2$), and each block has two nodes. One can see that each node in the block can be assigned different operations and can receive different inputs. The output of two nodes in the block can be combined by addition or concatenation operation; therefore, each block can be represented by a five-element tuple, i.e., (I_1, I_2, O_1, O_2, C), where $I_1, I_2 \in \mathcal{I}_b$ indicates the inputs to the block, $O_1, O_2 \in \mathcal{O}$ specifies the operations applied to inputs, and $C \in \mathcal{C}$ describes how to combine O_1 and O_2. As the blocks are ordered, the set of candidate inputs I_b for nodes in block b_k contains the output of the previous two cells and the set of the outputs of all previous blocks $\{b_i, i < k\}$ of the same cell. The first two inputs of the first cell of the whole model are set to the image data by default.

In the actual implementation, there are essential details to be noted. First, the number of channels may be different 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 usually uses 1×1 convolution filters such that it will not change the size of the input tensor. Second, as mentioned above, the input of a node in a block can come 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 input/output resolutions are different, the calibration operation has stride 2, otherwise 1. Besides, all the blocks have stride 1.

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

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

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

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

Obviously, the complexity of searching for the entire structure grows exponentially with the number of layers. For an intuitive comparison, we set the values for the variables in the Equation 2 and 3 the typical value in the literature,

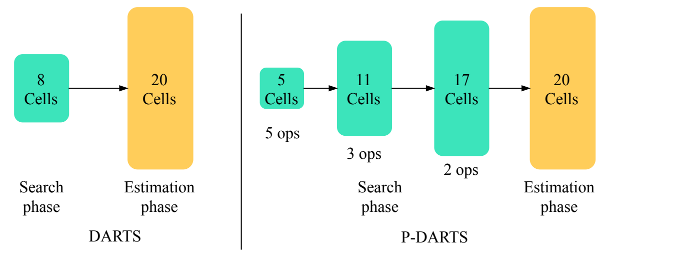


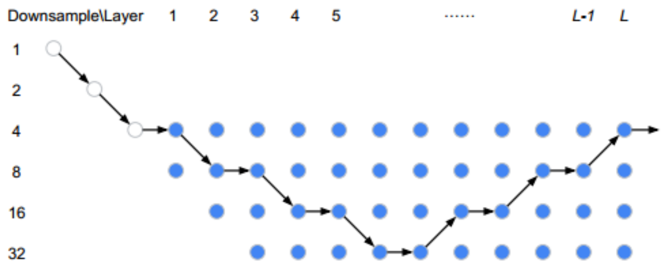
Figure 9: Difference between DARTS [17] and P-DARTS [125]. Both methods search and evaluate networks on the CIFAR-10 dataset. As the number of cell structures increases from 5 to 11 and 17, the number of candidate operations is gradually reduced accordingly.

i.e. $M = 5, L = 10, B = 3$, then $N_{entire} = 3.44 \times 10^{20}$ is much larger than $N_{cell} = 5.06 \times 10^{16}$.

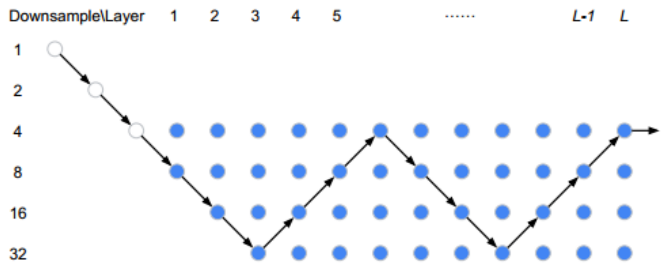
Two-stage Gap. Notably, the NAS methods of cell-based search space usually consist of two phases: search and estimation. Specifically, the best-performing model in the search phase is firstly selected, and then in the estimation phase, this model is trained from scratch or fine-tuned. However, there exists a large gap in the model depth between the two phases. As Figure 9 (left) shows, for DARTS [17], the generated model in the search phase only consists of 8 cells for reducing the consumption of GPU memory, while in the estimation phase, the number of cells is extended to 20. Although the search phase finds the best cell structure for the shallow model, it does not mean that it is still suitable for the deeper model in the evaluation phase. In other words, simply adding more cells may harm the model performance. To bridge this gap, Chen et al. [125] propose an improved method based on DARTS, namely Progressive DARTS (P-DARTS), which divides the search phase into multiple stages and increases the depth of the searched networks gradually at the end of each stage, hence bridging the gap between search and evaluation. However, increasing the number of cells in the search phase may bring an issue of heavier computational overheads. Thus, for reducing computational consumption, P-DARTS gradually reduces the number of candidate operations from 5 to 3 and 2 via search space approximation methods, shown in Figure 9. Experimentally, P-DARTS obtains a 2.50% error rate on the CIFAR-10 test dataset, outperforming the 2.83% error rate of 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, those approaches focus on the cell level and ignore the network level. As shown in Figure 8, whenever a fixed number of normal cells are stacked, the spatial dimension of feature maps is halved by adding a reduction cell. To jointly learn a good combination of repeatable cell and network



(a) Network-level architecture used in Conv-Deconv.

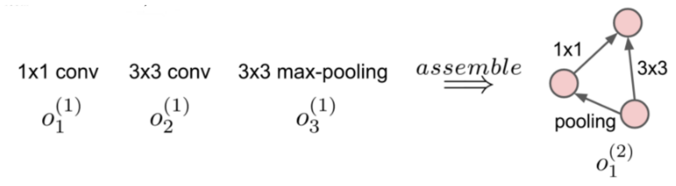


(b) Network-level architecture used in stacked hourglass networks.

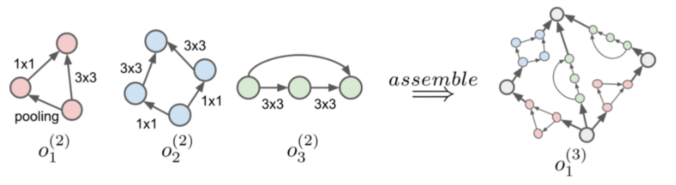
Figure 10: The network-level search space (the figure is adopted from[126]). Three gray nodes at the beginning indicate the fixed "stem" structures, and each blue point is a cell structure, as described above. The black arrows along the blue points indicate the final selected network-level structure.

structures, Liu et al. [126] define a general formulation for a network-level structure, depicted in Figure 10, from which many existing good network designs can be reproduced. In this way, we can fully explore the different number of channels and the size of feature maps of each layer in the network.

In terms of cell-level, the number of blocks (B) in a cell is still manually predefined and is fixed in the search stage. In other words, B is a new hyper-parameter that requires tuning by human input. To address this problem, Liu et al. [19] propose a novel hierarchical genetic representation scheme, namely *HierNAS*, in which the higher-level cell is generated by iteratively incorporating lower-level cells. As shown in Figure 11, the level-one cells can be some primitive operations, such as 1×1 and 3×3 convolution and 3×3 max-pooling. The level-one cells are the basic components of the level-two cells. Then, the level-two cells are used as the primitive operations to generate the 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 $G_{ij} = k$ means that the k-th operation 0_k is implemented between nodes i and j. For example, the level-two cell in Figure 11(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 discover more types of cell structures with more complex and flexible topologies. Similarly, Liu and Zoph et al. [18] also propose *Progressive NAS (PNAS)* to



(a) The level-one primitive operations are assembled into level-two cells.



(b) The level-two cells are viewed as primitive operations and assembled into level-three cells.

Figure 11: An example of a three-level hierarchical architecture representation. The figure is adopted from [19].

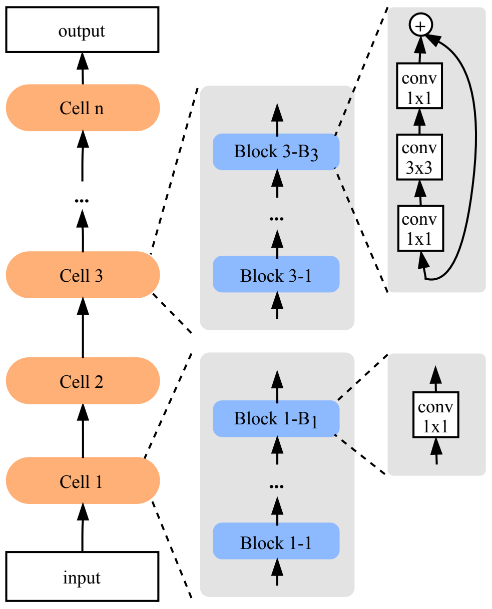


Figure 12: The factorized hierarchical search space in MnasNet [127]. The final network consists of different cells. The cell is composed of a variable number of repeated blocks, where the block in the same cell shares the same structure, but differs from the block in other cells.

search for the cell progressively, starting from the simplest cell structure that 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 layers of the network, which limits the layer diversity. Besides, many searched cell structures are complex and fragmented, which is critical for achieving both high accuracy and lower latency [127, 128]. To ease both problems, Tan et al. [127] propose MnasNet, which uses novel factorized hierarchical search space to generate different cell structures, namely MBConv, for different layers of the final network. Figure 12 presents the factorized hierarchical search space of MnasNet. The network consists of a predefined number of cell structures. Each cell has a different structure and contains a variable number of blocks, where each block has the same structure in the same cell but differs from the blocks in other cells. Since a good balance can be achieved between model performance and latency, many subsequent works [128, 129] also refer to this design method. Notably, due to the large computational consumption, most of the differentiable NAS (DNAS) works (e.g., DARTS) firstly search for a good cell structure on a proxy dataset (e.g., CIFAR10), then transfer it to the larger target dataset (e.g., ImageNet). Han et al. [129] propose ProxylessNAS that can directly search for neural networks on the targeted dataset and hardware platforms by using BinaryConnect [130], which addresses the high memory consumption issue.

4.1.4. Morphism-Based Search Space

Isaac Newton is reputed 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 [131] and transfer learning [132]. However, these methods do not directly modify the model structure. To this end, Chen et al. [20] propose the Net2Net technique to design new neural networks based on the existing network by inserting the identity morphism (IdMorph) transformations between neural network layers. IdMorph transformation is function-preserving and has two types: depth and width IdMorph, 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 width and depth separately, and 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 of the knowledge from its well-trained parent network, and continue to grow into a more robust network in a shortened training time. Specifically, compared with Net2Net, network morphism has the following advantages: 1) it can embed non-identity layers and handle arbitrary non-linear activation functions; 2) it can simultaneously perform depth, width, and kernel size-morphing in a single operation, whereas Net2Net can only separately consider depth and width changes. Experimental results show that network morphism can significantly accelerate the training process, as it uses one-fifteenth of the training time and achieves better results than the original VGG16.

Several subsequent research works [27, 22, 133, 134, 135, 136, 137, 138] are based on network morphism. For instance, Jin et al. [22] propose a framework that enables Bayesian optimization to guide the network morphism for

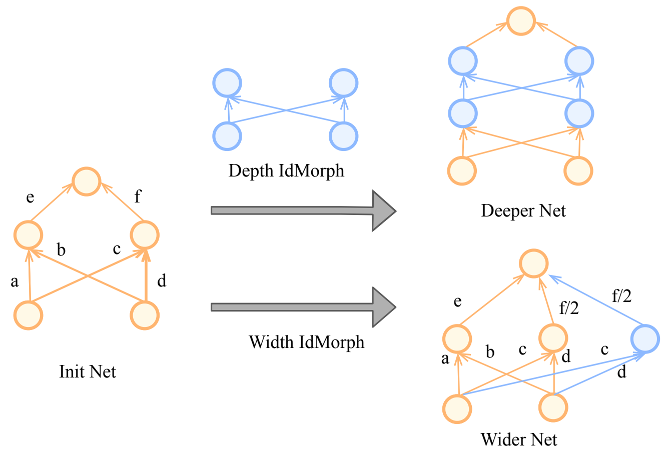


Figure 13: Net2DeeperNet and Net2WiderNet transformations in [20]. "IdMorph" indicates identity morphism operation.

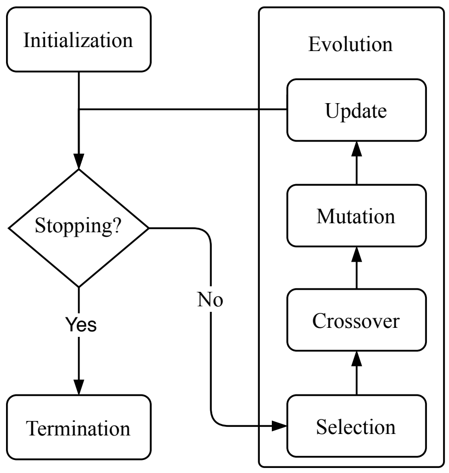


Figure 14: The overview of evolutionary algorithm.

an efficient neural architecture search. Wei et al [133] further improve network morphism at a higher level, i.e., by morphing a convolutional layer into an arbitrary module of a neural network. Additionally, Tan and Le [139] propose EfficientNet, which re-examines the effect of model scaling for convolutional neural networks and proves that carefully balancing 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 hyper-parameters that are tuned based on the performance observed on the validation set. However, this process highly depends on human experts and requires a lot of time and resources for trial and error. Therefore, many AO methods are proposed to free humans from this tedious procedure and to search for novel architectures automatically. We detail the commonly used AO methods in the following.

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, an evolutionary algorithm is a mature global optimization method with high robustness and broad applicability. It can effectively deal with 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 square of the number of nodes, and the number of nodes is fixed-length, i.e., predefined manually. To represent variable-length neural networks, the encoding of direct acyclic graph (DAG) becomes a promising solution [28, 25, 19]. For example, Suganuma et al. [28] use Cartesian genetic programming (CGP) [140, 141] encoding scheme to represent the 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, where 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. NeuroEvolution of Augmenting Topologies (NEAT) [24, 25] also uses a direct encoding scheme, where every node and every connection are stored. Indirect encoding specifies a generation rule to build the network and allows for a more compact representation. Cellular encoding (CE) [142] is an example of a system that utilizes indirect encoding of network structures. CE encodes a family of neural networks into a set of labeled trees and is based on a simple graph grammar. Recently, some studies [143, 144, 145, 27] describe the use of indirect encoding schemes to represent the network. For example, the network in [27] is encoded by function, and each network can be modified using function-preserving network morphism operators. Hence the capacity of the child network is increased and is guaranteed to perform at least as good as the parent networks.

Four Steps. A typical evolutionary algorithm consists of the following steps: selection, crossover, mutation, and update (see Figure 14):

- **Selection**: This step involves selecting a portion of the networks from all the generated networks for the crossover, which aims to maintain well-performing neural architectures while eliminating weak ones. There are three strategies for selecting networks. 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 i-th network. The second is *rank selection*, which is similar to fitness selection, but with a 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]. 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 that 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 via one-point or multiple-point crossover. However, the crossover of data arranged in such a fashion can sometimes lead to data damage. Thus Xie et al. [30] denote the basic unit in the crossover as a stage rather a bit, where a stage 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, allowing NEAT to add a new structure without losing track of which gene is throughout a 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 consists of randomly and independently flipping each bit. Two types of mutations are 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. In another approach, Real and Moore et al. [25] predefine a set of mutation operators, such as those that alter the learning rate and filter size and remove skin connections between nodes. By analogy with the biological process, although a mutation may look like a mistake that causes damage to the network structure and leads to a loss of functionality, the mutation also enables exploration of more novel structures and ensures diversity.

- **Update** Many new networks are generated by completing the above steps, and in light of 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] do not remove any networks from the population; instead, they allow the network to grow with time. EENA [146] regulates the population number via a variable λ, i.e., removing the worst model with probability λ and the oldest one with $1 - \lambda$.

4.2.2. Reinforcement Learning

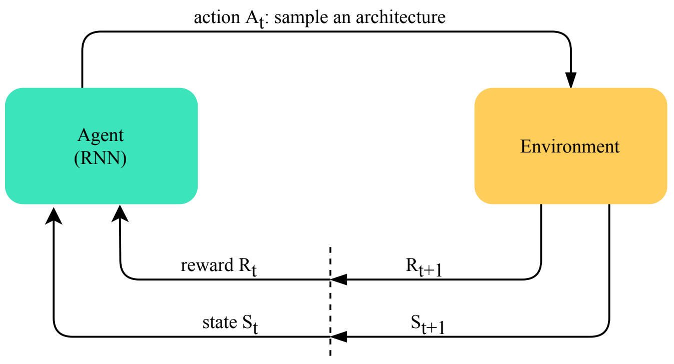


Figure 15: An overview of NAS using reinforcement learning.

Zoph et al. [12] were one of the first to apply reinforcement learning (RL) for neural architecture search. Figure 15 presents an overview of an RL-based NAS algorithm. The agent 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 agent's sampling strategy. Environment refers to the use of standard neural network training procedure to train and evaluate the network generated by the agent, after which the corresponding results (such as accuracy) are returned. Many follow-up approaches [23, 15, 16, 13] use this framework, but with different agent policies and neural-architecture encoding. Zoph et al. [12] first use the policy gradient algorithm [147] to train the agent, and sequentially sample a string to encode the entire neural architecture. In subsequent work [15], they use the Proximal Policy Optimization (PPO) algorithm [148] to update the agent, and propose the method shown in Figure 16 to build the cell-based neural architecture. MetaQNN [23] proposes a meta-modeling algorithm using Q-learning and a ϵ-greedy exploration strategy and experience replay to search neural architectures sequentially.

Although the above RL-based algorithms have achieved SOTA results on CIFAR-10 and Penn Treebank (PTB) [149] datasets, they spend much time and computational resources. For instance, [12] took 28 days and 800 K40 GPUs to search for the best-performing architecture, and

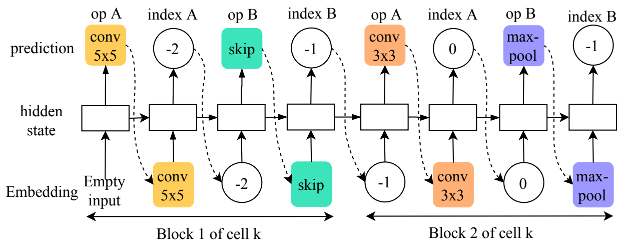


Figure 16: An example of an agent generating a cell structure. Each block in the cell consists of two nodes that are specified with different operations and inputs. The index −2 and −1 indicates the input comes from prev-previous and previous cell, respectively.

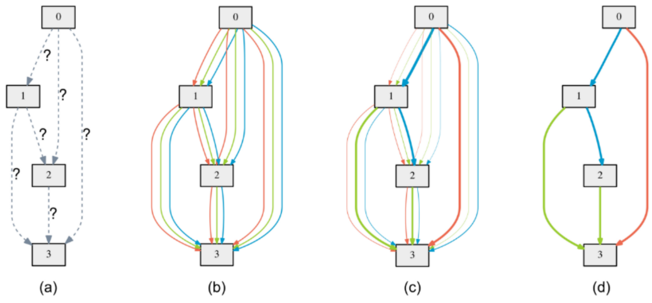


Figure 17: An overview of DARTS (the figure is adopted from [17]). (a) The data can only flow from lower-level nodes to higher-level nodes, and the operations on edges are initially unknown. (b) The initial operation on each edge is a mixture of candidate operations, each having equal weight. (c) The weight of each operation is learnable and ranges from 0 to 1, but for previous discrete sampling methods, the weight can only be 0 or 1. (d) The final neural architecture is constructed by preserving the maximum weight-value operation on each edge.

MetaQNN [23] also took ten days and 10 GPUs to complete its search, which is unaffordable for individual researchers, and even for some companies. To this end, some improved RL-based algorithms are discussed. BlockQNN [16] proposes a distributed asynchronous framework and an early-stop strategy to complete searching on only one GPU within 20 hours. The efficient neural architecture search (ENAS) [13] is even better, as it adopts a parameter-sharing strategy, in which all of the child architectures are regarded as sub-graph of a super-net; this enables these architectures to share parameters, obviating the need to train each child model from scratch. As a result, ENAS took only approximately 10 hours 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 search strategies above sample neural architectures from a discrete search space. A pioneering algorithm, namely DARTS [17], propose to search for neural architectures over the continuous and differentiable search space by using a softmax function to relax the discrete space, as outlined below:

$$\bar{o}_{i,j}(x)=\sum_{k=0}^{K}\frac{\exp\left(\alpha_{i,j}^{k}\right)}{\sum_{l=0}^{K}\exp\left(\alpha_{i,j}^{l}\right)}o^{k}(x) \tag{4}$$

where $o(x)$ indicates the operation performed on input x, $\alpha_{i,j}^{k}$ indicates the weight for 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 in an alternative way, suggestive of a bilevel optimization problem. Specifically, α and θ are optimized with the validation and the training set, respectively. The training and the validation loss are denoted by $\mathcal{L}_{train}$ and $\mathcal{L}_{val}$, respectively. Hence, the total loss function can be derived:

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

Figure 17 presents an overview of DARTS, where the cell is composed of N (here $N=4$) ordered nodes, and the node z^k (k starts from 0) is connected to the node $z^i, i\in\{k+1,...,N\}$. 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 super-net, which contains all possible child neural architectures. At the end of the search, the final architecture is derived only by retaining the maximum-weight operation for all mixed operations.

Although DARTS greatly reduces the search time, it has several problems. Firstly, as Eq. 5 shows, DARTS describes jointly optimization of the neural architecture and the 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. Single-level optimization is another solution and is 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 efficiently solved as the regular training, the searched architecture α commonly overfits on the training set and its performance on the validation set may not be guaranteed. In [150], the authors propose 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 weights of this neural architecture, and λ is a non-negative regularization variable that the weight of the training loss and validation loss. Specifically, when $\lambda=0$, Eq. 7 degrades to the single-level optimization (Eq. 6); while if λ is close to infinity, then Eq. 7 becomes a bilevel optimization (Eq. 5). The experimental results in [150] show that mixed-level optimization not only overcomes the overfitting issue of single-level optimization, but also avoids the gradient error of bilevel optimization.

Secondly, in DARTS, the output of each edge is the weighted sum of all candidate operation (shown in Eq. 4) during the whole search stage, which leads to a linear increase of the requirements of GPU memory with the number of candidate operations. To reduce resource consumption, many subsequent studies [151, 152, 150, 153, 128] develop a differentiable sampler to sample a child architecture from the super-net by using a reparameterization trick, namely Gumbel Softmax [154]. Specifically, the neural architecture is fully factorized and modeled with a concrete distribution [155], which provides an efficient way to sample a child architecture and allows gradient back-propagation. Therefore, Eq. 4 is re-formulated as Eq. 8:

$$\overline{o}_{i,j}^{k}(x)=\sum_{k=0}^{K}\frac{\exp\left(\left(\log\alpha_{i,j}^{k}+G_{i,j}^{k}\right)/\tau\right)}{\sum_{l=0}^{K}\exp\left(\left(\log\alpha_{i,j}^{l}+G_{i,j}^{l}\right)/\tau\right)}o^{k}(x) \tag{8}$$

where $G_{i,j}^{k}=-log(-log(u_{i,j}^{k}))$ is the k-th Gumbel sample, $u_{i,j}^{k}$ is a uniform random variable, and τ is the softmax temperature. When $\tau\rightarrow\infty$, the possibility distribution of all operations between each pair of nodes approximates to the one-hot distribution. In GDAS [151], only the operation with the maximum possibility for each edge is selected during the forward pass, while the gradient is back-propagated according to Eq. 8. In other words, only one path of the super-net is selected for training, thereby reducing GPU memory usage. On the other hand, ProxylessNAS [129] alleviates the huge consumption of resources via path binarization. Specifically, ProxylessNAS transforms the real-valued path weights [17] to binary gates, which activates only one path of the mixed operations and hence solve the memory issue.

Another problem is that optimizing different operations together is difficult, as those operations may compete with each other, leading to a negative influence. For example, several studies [156, 125] find that skip-connect operation will dominate 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+ [156] 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, the solution in P-DARTS [125] is to regularize the search space, which executes 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 method is surrogate model-based optimization (SMBO) algorithms [33, 34, 157, 158, 159, 160, 161, 162, 163, 18, 158]. The core 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. In this way, these methods can greatly shorten the time for searching and thus improve efficiency.

SMBO algorithms differ from the surrogate models, which can be broadly divided into Bayesian optimization methods (including *Gaussian process (GP)* [164], *random forests (RF)* [37], *Tree-structured Parzen Estimator (TPE)* [165]), and neural networks [161, 166, 18, 163].

Bayesian optimization (BO) [167, 168] is one of the most popular and well-established methods for hyper-parameter optimization. Recently, many follow-up works [33, 34, 157, 158, 159, 160, 161, 162] make efforts to apply those SOTA BO methods to architecture optimization. For example, In [169, 170, 157, 162, 171, 172], the validation results of the generated neural architectures are modeled as a Gaussian process, which guides to search for the optimal neural architectures. However, GP-based Bayesian optimization methods suffer from that the inference time scales cubically in the number of observations, and they are not good at dealing with variable-length neural networks. Camero et al. [173] propose three fixed-length encoding schemes to cope with variable-length problems by employing a random forest as the surrogate model. Similarly, both [33] and [173] also use a random forest as the surrogate model, and [174] shows that random forest works better in the setting of high dimensionality than GP-based methods.

Instead of using BO, some studies use a neural network as the surrogate model. For example, in PNAS [18] and EPNAS [163], an LSTM is derived as the surrogate model to predict variable-sized architectures progressively. At the same time, NAO [166] uses a simpler surrogate model, i.e., Multilayer Perceptron (MLP), and NAO is more efficient and achieves better results on CIFAR-10 compared with PNAS [18]. White et al. [161] train an ensemble of neural networks to predict the mean and variance of 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 and have also been applied to several NAS studies [175, 176, 177, 11]. For instance, in MnasNet [127], the number of cells and blocks are predefined manually. To further automate the NAS process, Geifman et al. [176] propose a modular architecture search space ($\mathcal{A}=\{A(B,i,j)|i\in\{1,2,...,N_{cells}\},j\in\{1,2,...,N_{blocks}\}\}$) 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. Obviously, the bigger $N_{cells}\times N_{blocks}$ is set, the larger space is explored, but the more resources are also required.

In [177], the authors conduct an effectiveness comparison between SOTA NAS methods and random search. The results show that random search is a competitive NAS baseline. Specifically, random search with early-stopping strategy performs as well as ENAS [13], which is a reinforcement learning-based leading NAS method. Besides, the work by Yu et al. [11] also demonstrate that the SOTA

NAS techniques are not significantly better than random search.

4.2.6. Hybrid Optimization Method

The architecture optimization methods mentioned above have their own advantages and disadvantages. 1) The evolutionary algorithm (EA) is a mature global optimization method with high robustness. However, EA requires a lot of computational resources [26, 25], and its evolution operations (such as crossover and mutations) are performed randomly. 2) Although the RL-based methods (e.g., ENAS [13]) can learn complex architectural patterns, the searching efficiency and stability of the RL agent are not guaranteed, because the RL agent needs to try amounts of actions to get a positive reward. 3) The gradient descent-based methods (e.g., DARTS [17]) greatly improve searching efficiency by relaxing the categorical candidate operations to continuous variables. Nevertheless, in essence, they all search for a child network from a super network, which limits the diversity of neural architectures. Therefore, some methods propose to incorporate different optimization methods to capture the best of their worlds, and we summarize these methods as follows.

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

EA+GD. Yang et al. [40] combine the evolutionary algorithm and gradient descent-based method. The architectures share parameters within one super network and are tuned on the training set with a few epochs. Then the populations and the super network are directly inherited in the next generation, which greatly accelerates the evolution. [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. In [43], the authors use the random forest as a surrogate to predict model performance, which accelerates the fitness evaluation in an evolutionary algorithm.

GD+SMBO. Different from DARTS, which learns weights for candidate operations, NAO [166] proposes a variational auto-encoder to generate neural architectures and further build a regression model as a surrogate to predict the performance of a generated architecture. Specifically, the encoder maps the representations of the neural architecture to continuous space, 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.

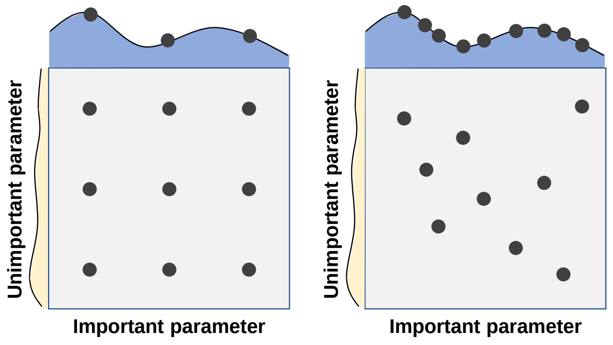


Figure 18: Examples of grid search (left) and random search (right) in nine trials for optimizing a two-dimensional space function $f(x, y) = g(x) + h(y) \approx g(x)$ [178]. The parameter in $g(x)$ (light-blue part) is relatively important, but the parameter in $h(y)$ (light-yellow part) is unimportant. In a grid search, nine trials cover only three different 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. (the figure is adopted from [178])

4.3. Hyper-parameter Optimization

Most NAS methods use the same set of hyper-parameters for all candidate architectures during the whole search stage, so after finding the most promising neural architecture, it is necessary to redesign a set of hyperparameters, and use this set of hyperparameters to retrain or fine-tune the architecture. Since some HPO methods (e.g., Bayesian optimization, and evolutionary optimization) have also been applied in NAS, thus we will only briefly introduce these methods. The commonly used hyper-parameter optimization (HPO) methods include grid search, random search, Bayesian optimization, gradient-based optimization, evolutionary optimization, and population-based optimization.

4.3.1. Grid and Random Search

Figure 18 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; in contrast, RS, as its name suggests, selects the best point from a set of points drawn at random.

GS is very simple and naturally supports parallel implementation, but it is computationally expensive and inefficient when the hyper-parameter space is very large, as the number of trials grows exponentially with the dimensionality of hyper-parameters. To alleviate this problem, Hsu et al. [179] propose coarse-to-fine grid search, i.e., first to inspect a coarse grid to locate a good region, then implement a finer grid search on the identified region. Similarly, Hesterman et al. [180] propose a contracting-grid search 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 separation of points in the new grid is reduced to half of that on the old grid.

Iterations of the above procedure are performed until the results converge to a local minimum.

Although [178] empirically and theoretically shows that random search is more practical and efficient than grid search, random search does not promise an optimum. This means that although the longer the search, the more likely it is to find the optimal hyper-parameters, it will consume more resources. Li and Jamieson et al. [181] propose *hyperband* algorithm to make a trade-off between the performance of hyper-parameters and resource budgets. Hyperband allocates limited resources (such as time or CPUs) to only the most promising hyper-parameters, by successively discarding the worst half of configuration settings long before the training process is finished.

4.3.2. Bayesian Optimization

Bayesian optimization (BO) is an efficient method for global optimization of expensive blackbox functions. In this part, we will only provide a brief introduction to BO. For an in-depth discussion for BO, we recommend readers to refer to these excellent surveys [168, 167, 182, 183].

BO is a surrogate model-based optimization (SMBO) method, which builds a probabilistic model mapping from hyperparameters to the objective metrics evaluated on the validation set. It well balances exploration (evaluating as many sets of hyperparameters as possible) and exploitation (allocating more resources to those promising hyperparameters).

Algorithm 1 Sequential Model-Based Optimization

```
INPUT: $f, \Theta, S, \mathcal{M}$
$\mathcal{D} \leftarrow$ INITSAMPLES $(f, \Theta)$
**for** $i$ in $[1, 2, .., T]$ **do**
    $p(y|\theta, \mathcal{D}) \leftarrow$ FITMODEL $(\mathcal{M}, \mathcal{D})$
    $\theta_i \leftarrow \arg\max_{\theta \in \Theta} S(\theta, p(y|\theta, \mathcal{D}))$
    $y_i \leftarrow f(\theta_i)$    $\triangleright$ 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 [167]), which presents that several inputs need to be predefined at initial, including evaluation function f, search space Θ, acquisition function $\mathcal{S}$, probabilistic model $\mathcal{M}$, and the records 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 is the evaluation result of the sampled neural architecture. After the initialization, the steps of SMBO are as follows:

1. The first step is to tune the probabilistic model $\mathcal{M}$ to fit the records dataset D.
2. The acquisition function $\mathcal{S}$ is used to select the next promising neural architecture from the probabilistic model $\mathcal{M}$.
3. After that, the performance of the selected neural architecture will be 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 records dataset D is updated by appending a new pair of result (θ_i, y_i).

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

Table 2: The open-source Bayesian optimization libraries. GP, RF, and TPE represent *Gaussian process* [164], *random forests* [37], *Tree-structured Parzen Estimator* [165], respectively.

The above four steps are repeated for T times, where T needs to be specified according to the total time or resources available. The commonly used surrogate models for BO method are Gaussion process (GP), random forest (RF), and Tree-structured Parzen Estimator (TPE). Table 2 summarize existing open-source BO methods, where GP is one of the most popular surrogate model. 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] propose the BO-based Hyperband (BOHB) algorithm, which combines the strengths of TPE-based BO and Hyperband, and hence perform much better than standard BO methods. Furthermore, FABOLAS [35] is a faster BO procedure, which maps the validation loss and training time as a function of dataset size, i.e., trains a generative model on a sub-dataset that gradually increases in size. As a result, FABOLAS is 10 to 100 times faster than other SOTA BO algorithms and identifies the most promising hyper-parameters.

4.3.3. Gradient-Based Optimization

Another group of HPO methods are *gradient-based optimization (GO)* algorithms [184, 185, 186, 187, 188, 189]. Different to above blackbox HPO methods (e.g, GS,RS, and BO), GO methods use the gradient information to optimize hyperparameters, and greatly improve the efficiency of HPO. Maclaurin et al. [186] propose a reversible-dynamics

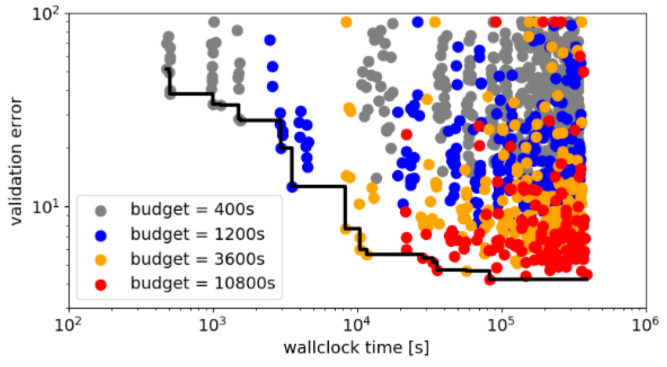


Figure 19: Validation error of each configuration generated on different budgets during the whole search procedure (The figure is adopted from [34]). Differently colored points have different time budgets for the search. For example, the time budget of grey points is 400 s. Neural networks with a time budget of 3,600 s (orange points) even perform similarly to those with a budget of 10,800 s (red points).

memory-tape approach to handle thousands of hyperparameters efficiently through gradient information. However, optimizing many hyperparameters is computationally challenging. To alleviate this issue, [187] use approximate gradient information rather than true gradient to optimize continuous hyperparameters, in which case the hyperparameters can be updated before the model is trained to converge. Franceschi et al. [188] study both the reverse-mode and forward-mode GO methods. The reverse-mode method differs from [186] and does not require reversible dynamics, whereas it needs to store the entire training history for computing the gradient with respect to hyperparameters. The forward-mode method overcomes this problem by real-time updating hyperparameters, and is demonstrated to significantly improve the efficiency of HPO on large datasets. Chandra [189] propose the gradient-based ultimate optimizer, which can not only optimize the regular hyperparameters (e.g., learning rate), but also optimize the hyperparameters of optimizer (e.g., Adam optimizer [190]'s moment coefficient β_1, β_2).

5. Model Estimation

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. Several algorithms have been proposed for accelerating the process of model evaluation, and are summarized as follows.

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 estimation. In [191], ImageNet64×64 and its variants 32×32, 16×16 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, [192] 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 demonstrate 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 (see Figure 19).

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. [193] propose 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 [194] uses a single-path over-parameterized ConvNet to encode all architectural decisions with shared convolutional kernel parameters.

5.3. Surrogate

The surrogate-based method [195, 196, 197, 43] is another powerful tool that approximates to the black-box function. 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 control the method of searching. 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 in terms of total computational speed. However, when the optimization space is too large and hard to quantify, and the evaluation of each configuration is extremely expensive [198], a surrogate-based method is not applicable. Luo et al. [199] propose *SemiNAS*, a semi-supervised NAS method, which leverages amounts of unlabeled architectures to further improve searching efficiency. The accuracy of the generated model does not need to be evaluated after training the model, but can be obtained only by using the controller to predict the accuracy.

5.4. Early stopping

Early stopping was first used to prevent overfitting in classical ML. It is used in several recent studies [200, 201, 202] to accelerate model evaluation by stopping evaluations that are predicted to perform poorly on the validation set. For example, [202] 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, [203] 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.

5.5. Resource-aware

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 [204] 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 [127] proposes a customized weighted product to approximate a Pareto optimal solution:

$$\underset{m}{\operatorname{maximize}} \quad ACC(m) \times \left[\frac{LAT(m)}{T}\right]^{w} \tag{9}$$

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

$$w = \begin{cases} \alpha, & \text{if } LAT(m) \leq T \\ \beta, & \text{otherwise} \end{cases} \tag{10}$$

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

In terms of a differentiable neural architecture search (DNAS) framework, the constraint (i.e., loss function) should be differentiable. To do so, FBNet [128] 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:

$$\mathcal{L}\left(a, \theta_{a}\right) = \mathrm{CE}\left(a, \theta_{a}\right) \cdot \alpha \log(\mathrm{LAT}(a))^{\beta} \tag{11}$$

where $CE(a, \theta_a)$ indicates the cross-entropy loss of architecture a with weights θ_a. Like MnasNet [127], this loss function has two hyper-parameters that need to be set manually: α and β control the magnitude of the loss function and the latency term, respectively. In SNAS [152], 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.

6. NAS Performance Summary

In Section 4, we review the different types of search space and architecture optimization methods. We also summarize commonly used model estimation methods in Section 5. These two sections introduce a lot of NAS studies, which may cause the readers to get lost in details. Therefore, in this section, we will summarize and compare the performance of these NAS algorithms from a global perspective to give readers a clearer and more comprehensive understanding of the development of NAS methods. Then we will discuss some worth studying problems of NAS technique and introduce NAS applications beyond image classification.

6.1. NAS performance comparison

Notably, many NAS studies may propose several neural architecture variants, where each variant is designed for different scenarios (e.g., some architectures perform better but with large size, while some are lightweight for a mobile device but with a performance penalty), so we only report the representative results for each work. Besides, to ensure a valid comparison, we take the accuracy and the algorithm efficiency as comparison indices. As the number and types of GPUs used vary from different studies, we use *GPU Days* to approximate the efficiency, which is defined as:

$$\text{GPU Days} = N \times D \tag{12}$$

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

Table 3 and Table 4 present the performance of different NAS studies on CIFAR-10 and ImageNet, respectively. Besides, since most NAS methods first search for the neural architecture based on a small dataset (CIFAR-10), then transfer the architecture to a larger dataset (ImageNet), the search time for both datasets may be the same. It can be seen from the table that the early work of EA and RL based NAS methods pay more attention to high performance, regardless of the resource consumption. For example, although AmoebaNet [26] achieved excellent results on both CIFAR-10 and ImageNet, it took 3,150 GPU days with 450 GPUs for searching. The subsequent NAS studies make efforts to improve searching efficiency while ensuring the searched model's high performance. For instance, EENA [146] elaborately designs the mutation and crossover operations, which can reuse the learned information to guide the evolution process and hence greatly improve the efficiency of EA-based NAS method. ENAS [13] is one of the first RL-based NAS methods to propose the parameter-sharing strategy, which reduces the number

Reference	Published in	#Params (Millions)	Top-1 Acc(%)	GPU Days	#GPUs	AO
ResNet-110 [2]	ECCV16	1.7	93.57	-	-	Maunally designed
PyramidNet [205]	CVPR17	26	96.69	-	-	
DenseNet [124]	CVPR17	25.6	96.54	-	-	
GeNet#2 (G-50) [30]	ICCV17	-	92.9	17	-	EA
Large-scale ensemble [25]	ICML17	40.4	95.6	2,500	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	3,150	450 K40	
AmoebaNet-B (N=6, F=36)+c/o [26]	AAAI19	2.8	97.45	3,150	450 K40	
Lemonade [27]	ICLR19	3.4	97.6	56	8 Titan	
EENA [146]	ICCV19	8.47	97.44	0.65	1 Titan Xp	
EENA (more channels)[146]	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	2,000	500 P100	
NASNet-A (6 @ 768)+c/o [15]	CVPR18	3.3	97.35	2,000	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 [136]	ICML18	5.7	97.01	200	-	
Path-level EAS+c/o [136]	ICML18	5.7	97.51	200	-	
ProxylessNAS-RL+c/o[129]	ICLR19	5.8	97.70	-	-	
FPNAS[206]	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 [175]	ArXiv19	3.6	98.07	0.8	1 2080Ti	
P-DARTS+c/o[125]	ICCV19	3.4	97.50	0.3	-	
P-DARTS(large)+c/o[125]	ICCV19	10.5	97.75	0.3	-	
SETN[207]	ICCV19	4.6	97.31	1.8	-	
GDAS+c/o [151]	CVPR19	2.5	97.18	0.17	1	
SNAS+moderate constraint+c/o [152]	ICLR19	2.8	97.15	1.5	1	
BayesNAS[208]	ICML19	3.4	97.59	0.1	1	
ProxylessNAS-GD+c/o[129]	ICLR19	5.7	97.92	-	-	
PC-DARTS+c/o [209]	CVPR20	3.6	97.43	0.1	1 1080Ti	
MiLeNAS[150]	CVPR20	3.87	97.66	0.3	-	
SGAS[210]	CVPR20	3.8	97.61	0.25	1 1080Ti	
GDAS-NSAS[211]	CVPR20	3.54	97.27	0.4	-	
NASBOT[157]	NeurIPS18	-	91.31	1.7	-	SMBO
PNAS [18]	ECCV18	3.2	96.59	225	-	
EPNAS[163]	BMVC18	6.6	96.29	1.8	1	
GHN[212]	ICLR19	5.7	97.16	0.84	-	
NAO+random+c/o[166]	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 [177]	UAI19	4.3	97.15	2.7	-	
DARTS - random+c/o [17]	ICLR19	3.2	96.71	4	1	
RandomNAS-NSAS[211]	CVPR20	3.08	97.36	0.7	-	
NAO+weight sharing+c/o [166]	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

Table 3: The 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 [88]. RL, EA, GD, RS, BO 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	-	-	Maunally designed
PyramidNet [205]	CVPR17	116.4	70.8/95.3	-	-	
SENet-154 [123]	CVPR17	-	71.32/95.53	-	-	
DenseNet-201 [124]	CVPR17	76.35	78.54/94.46	-	-	
MobileNetV2 [213]	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	3,150	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	3,150	450 K40	
GreedyNAS [214]	CVPR20	6.5	77.1/93.3	1	-	
NASNet-A(4@1056)	ICLR17	5.3	74.0/91.6	2,000	500 P100	RL
NASNet-A(6@4032)	ICLR17	88.9	82.7/96.2	2,000	500 P100	
Block-QNN[16]	CVPR18	91	81.0/95.42	96	32 1080Ti	
Path-level EAS[136]	ICML18	-	74.6/91.9	8.3	-	
ProxylessNAS(GPU) [129]	ICLR19	-	75.1/92.5	8.3	-	
ProxylessNAS-RL(mobile) [129]	ICLR19	-	74.6/92.2	8.3	-	
MnasNet[127]	CVPR19	5.2	76.7/93.3	1,666	-	
EfficientNet-B0[139]	ICML19	5.3	77.3/93.5	-	-	
EfficientNet-B7[139]	ICML19	66	84.4/97.1	-	-	
FPNAS[206]	ICCV19	3.41	73.3/-	0.8	-	
DARTS (searched on CIFAR-10)[17]	ICLR19	4.7	73.3/81.3	4	-	GD
sharpDARTS[175]	Arxiv19	4.9	74.9/92.2	0.8	-	
P-DARTS[125]	ICCV19	4.9	75.6/92.6	0.3	-	
SETN[207]	ICCV19	5.4	74.3/92.0	1.8	-	
GDAS [151]	CVPR19	4.4	72.5/90.9	0.17	1	
SNAS[152]	ICLR19	4.3	72.7/90.8	1.5	-	
ProxylessNAS-G[129]	ICLR19	-	74.2/91.7	-	-	
BayesNAS[208]	ICML19	3.9	73.5/91.1	0.2	1	
FBNet[128]	CVPR19	5.5	74.9/-	216	-	
OFA[215]	ICLR20	7.7	77.3/-	-	-	
AtomNAS[216]	ICLR20	5.9	77.6/93.6	-	-	
MiLeNAS[150]	CVPR20	4.9	75.3/92.4	0.3	-	
DSNAS[217]	CVPR20	-	74.4/91.54	17.5	4 Titan X	
SGAS[210]	CVPR20	5.4	75.9/92.7	0.25	1 1080Ti	
PC-DARTS [209]	CVPR20	5.3	75.8/92.7	3.8	8 V100	
DenseNAS[218]	CVPR20	-	75.3/-	2.7	-	
FBNetV2-L1[219]	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[212]	ICLR19	6.1	73.0/91.3	0.84	-	
SemiNAS[199]	CVPR20	6.32	76.5/93.2	4	-	
Hierarchical-random[19]	ICLR18	-	79.6/94.7	8.3	200	RS
OFA-random[215]	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

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

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 greatly reduce computational resource consumption for searching and achieve SOTA results. A lot of follow-up work is to make further improvement and optimization in this direction. Interestingly, random search-based methods can also obtain comparable results. The authors of [177] demonstrate that random search with weight-sharing can outperform a series of powerful methods, such as ENAS [13] and DARTS [17].

6.1.1. Kendall Tau metric

Since random search is comparable to those more sophisticated methods (e.g., DARTS and ENAS), one natural question is: what are the advantages and significance of the other architecture optimization algorithms compared with the random search? The researchers try to use other metrics to answer this question, not just considering the final accuracy of the model. Most NAS methods consist of two stages: 1) search for a best-performing architecture on the training set; 2) then expand it to a deeper one 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 on the validation set. Therefore, instead of merely considering the final accuracy and search time cost, many NAS studies [217, 220, 211, 11, 120] make use of Kendall Tau (τ) metric [221] to measure the correlation of the model performance between the search and estimation stages. The value of τ is defined as:

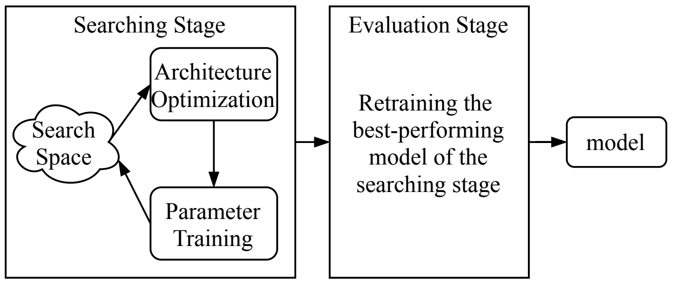
$$\tau = \frac{N_C - N_D}{N_C + N_D} \tag{13}$$

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

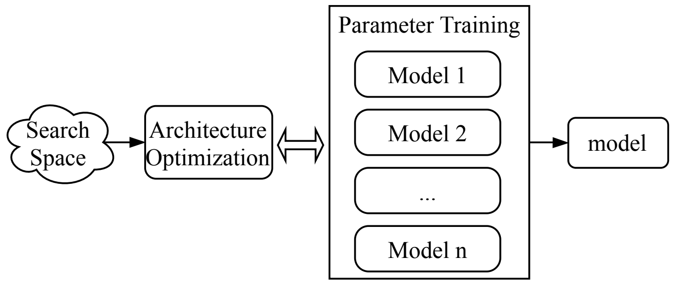
- $\tau = 1$: two rankings are identical
- $\tau = -1$: two rankings are completely opposite.
- $\tau = 0$: there is no relationship between two rankings.

6.1.2. NAS-Bench Dataset

Although Table 3 and Table 4 presents 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), data augmentation (e.g., Cutout [88]). In other words, the comparison is not quite fair. In this context, NAS-Bench-101 [222] is a pioneering work to alleviate the problem of non-reproducibility. It provides a tabular dataset that contains 423,624 unique neural networks, which are generated and evaluated from a fixed graph-based search space, and mapped to their trained and evaluated performance on CIFAR-10. Besides, Dong et al. [223] further build NAS-Bench-201, which is an extension to NAS-Bench-101 and has different search space, results on multiple datasets (CIFAR-10, CIFAR-100, and ImageNet-16-120 [191]), and more diagnostic information. Similarly, Klyuchnikov et al. [224] propose a NAS-Bench for the NLP task. These datasets enable NAS researchers to focus solely on verifying the effectiveness and efficiency of their proposed architecture optimization algorithms, avoiding repetitive training for selected architectures, and greatly helping the NAS community develop.


Searching Stage
Search Space
Architecture Optimization
Parameter Training
Evaluation Stage
Retraining the best-performing model of the searching stage
model

(a) Two-stage NAS consists of the searching stage and evaluation stage. The best-performing model of the searching stage will be 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 process of architecture optimization and parameter training run simultaneously.

Figure 20: Illustration of two-stage and one-stage NAS flow.

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 Figure 20.

Two-stage NAS consists of the *searching stage* and *evaluation stage*. The *searching stage* involves architecture optimization and parameter training. The simplest idea is to train all possible architectures from scratch and then choose the optimal one. However, it requires many resources (e.g., NAS-RL [12] took 22,400 GPU days with 800 K40 GPUs for searching) infeasible for most companies and institutes. Therefore, most NAS methods (such as ENAS [13] and DARTS [17]) sample and train a large number of candidate architectures in the searching stage, and then the best-performing one will be further retrained in the evaluation stage.

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 architecture optimization and parameter training simultaneously. In this way, it can greatly improve efficiency. However, it is currently the main challenge of any general NAS problem. Several recent studies [215, 225, 226, 216] have tried to overcome this challenge. For instance, in [215], the authors

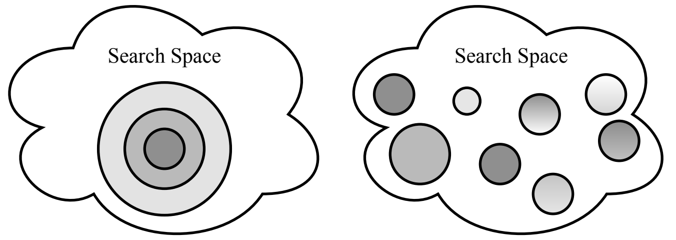


Figure 21: (Left) One-shot models. (Right) Non one-shot models. Each circle indicates a different model, and the area of the circle indicates the size of the model. We use concentric circles to represent one-shot models, as they share the weights with each other.

propose *progressive shrinking* algorithm to post-process the weights after training is finished. Specifically, they first pretrain the full neural network, then progressively fine-tune the smaller networks that share weights with the full network. Based on well-designed constraints, the performance of all sub-networks is guaranteed. Thus, given a target deployment device, a specialized sub-network can be directly exported without fine-tuning. However, [215] still requires many computational resources, as the whole process took 1,200 GPU hours with V100 GPUs. BigNAS [226] revisits conventional training techniques of stand-alone networks, and empirically proposes several techniques to handle a wider set of models, ranging in size from 200M FLOPs to 1G FLOPs, while [215] only handles models under 600M FLOPs. Both AtomNAS [216] and DSNAS [217] propose 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. We need to know that one-shot is not exactly equivalent to one-stage. As mentioned above, we divide NAS studies into one-stage and two-stage methods according to the flow (Figure 20), while whether a NAS algorithm belongs to the one-shot method or not depends on whether candidate architectures share the same weights (Figure 21). We observe that most of one-stage NAS methods are based on one-shot paradigm.

What is One-shot NAS? One-shot NAS methods embeds the search space into an over-parameterized supernet. As a result, all possible architectures can be derived from the supernet. Figure 20 shows the difference between the search space 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 among each other. One-shot NAS methods can be divided into two categories, which differ in how to handle architecture optimization and weights training: coupled and decoupled optimization [227, 214].

Coupled optimization. The first category of one-shot NAS methods optimizes the architecture and weights in a coupled manner [13, 17, 151, 129, 152]. For instance, ENAS [13] proposes to use 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 performance will be recorded. Finally, the best-performing architecture is selected for further retraining. DARTS [17] also uses a similar weight sharing strategy, but the architecture distribution is continuously parameterized. The supernet contains all candidate operations, each with learnable parameters. The best architecture can be directly derived from the distribution. However, since DARTS [17] directly optimizes the supernet weights and the architecture distribution, it suffers from the vast GPU memory consumption. Although DARTS-like methods [129, 151, 152] have proposed different approaches to reduce resource requirements, coupled optimization inevitably introduces a bias in both architecture distribution and supernet weights [194, 227], as they treat all sub-networks unequally. Specifically, the quickly-converged architectures can easily obtain more chances to be optimized [17, 156], and those architectures are only a small portion of the whole candidates; therefore, it is very harsh 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 will be negatively impacted, leading to performance degradation. [228] defines this phenomenon as *multi-model forgetting*. To overcome this problem, Zhang et al. [229] model the supernet training as a constrained optimization problem of continual learning, and propose *novel search based architecture selection* (NSAS) loss function. They apply the proposed method to RandomNAS [177] and GDAS [151], and the experimental result demonstrates that the method is effective to reduce the multi-model forgetting and boost the predictive ability of the supernet as an evaluator.

Decoupled optimization. The second category of one-shot NAS methods [207, 230, 227, 215] solves the problem of above one-shot methods by decoupling the optimization of architecture and weights into two sequential phases: 1) training the supernet; 2) using the trained supernet as a predictive performance estimator of different architectures to select the most promising architecture.

In terms of the phase of supernet training, the supernet cannot be directly trained like a regular neural network, because *the weights of the supernet are also deeply coupled* [194]. The work by Yu et al. [11] experimentally shows that weight sharing strategy harms the individual architecture's performance and negatively impacts the real performance ranking of the candidate architectures. To reduce weight coupling, many one-shot NAS methods [194, 207, 14, 212] propose the random sampling policy, which uniformly randomly samples an architecture from the supernet, and only the weights of this architecture are activated and optimized. RandomNAS [177] also demonstrates that a random search policy is a competitive baseline method. Although the strategy of sampling and training only one path of the supernet each time is already adopted in previous one-shot approaches [151, 13, 152, 129, 128], they sample the path according to RL controller [13], Gumbel Softmax [151, 152, 128], or BinaryConnect network [129], which instead highly couples the architecture and supernet weights. SMASH [14] proposes an auxiliary hypernetwork to generate the weights for randomly sampled architectures. Similarly, Zhang et al. [212] propose a computation graph representation, and use the graph hypernetwork (GHN) to predict the weights for all possible architectures faster and more accurately than regular hypernetworks (SMASH [14]). However, Bender et al. [230] conduct a careful experimental analysis to understand the mechanism of weight sharing strategy, and show that neither a hypernetwork nor an RL controller is required to find the optimal architecture. In [230], the authors propose a *path dropout* strategy to alleviate the problem of weight coupling. During training the supernet, each path of the supernet is randomly dropped with gradually increasing probability. GreedyNAS [214] propose a multi-path sampling strategy to train the greedy supernet. The strategy can focus on more potentially-good paths, and is demonstrated to be effective in achieving a fairly high rank correlation of candidate architectures compared with random search.

The second phase involves how to select the most promising architecture from the trained supernet, which is the primary purpose of most NAS tasks. Both SMASH [14] and [230] randomly select a set of architectures from the supernet, and rank 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 with the weights generated by the hypernetwork. Besides, [230] observe that the architectures with smaller symmetrized KL divergence value are more likely to perform better. The equation is as follows:

$$\begin{aligned} D_{\text{SKL}} &= D_{\text{KL}}(p\|q) + D_{\text{KL}}(q\|p) \\ s.t.\ D_{\text{KL}}(p\|q) &= \sum_{i=1}^{n} p_i \log \frac{p_i}{q_i} \end{aligned} \tag{14}$$

where $(p_1, ..., p_n)$ and $q_1, ..., q_n$ indicates the prediction of the sampled architecture and one-shot model, and n is the number of classes. The cost of calculating the KL value is very small; in [230], only 64 random training data examples are used. Evolutionary algorithm (EA) is also a promising search solution [194, 214]. For instance, SPOS [194] makes use of 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. Self-Evaluated Template Network (SETN) [207] proposes an *estimator* to predict the probability of each architecture being likely to have a lower validation loss. The experimental results show that SETN can potentially find the architecture with better performance than random search based methods [230, 14].

6.4. Joint Hyper-parameter and Architecture Optimization

Most NAS methods fix the same setting of training-related hyper-parameters during the whole search stage. After the search, the hyper-parameters of the best-performing architecture will be further optimized. However, this paradigm may result in sub-optimal results as different architectures tend to fit different hyper-parameters, making the ranking of the models unfair [231]. Therefore, a promising solution is the joint *hyper-parameter and architecture optimization (HAO)* [34, 232, 231, 233]. For instance, Zela et al. [34] casts NAS as a hyperparameter optimization problem. In this way, the search spaces of NAS and standard hyper-parameters are combined, and BOHB [38], an efficient HPO method, is applied to optimize the architecture and hyper-parameters jointly. Similarly, Dong et al. [231] propose 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 architecture (e.g., the number of layers) and hyper-parameters (e.g., learning rate). The challenge is that the candidate choices for architecture search space are usually categorical, while hyper-parameters 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 $\{0.1, 0.2, 0.3\}$, then the final learning rate is defined as $lr = w_1 \times 0.1 + w_2 \times 0.2 + w_3 \times 0.3$. FBNetv3 [233] also proposes to jointly search both architectures and the corresponding training recipes (i.e., hyper-parameters). 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 hyper-parameters is used to train the accuracy predictor, which will be applied to search for promising architectures and hyper-parameters at the later stage.

7. Open Problems and Future Work

In this section, we discuss several open problems of the existing AutoML methods, and propose some future directions for research.

7.1. Flexible Search Space

As summarized in Section 4, there are several types of search spaces, where the primitive operations can be roughly classified into pooling and convolution. Some even use a more complex module (MBConv [127]) as the primitive operation. Although these search spaces have been proven effective to generate well-performing neural architectures, they all based on human knowledge and experience, which inevitably introduces human bias and hence still does

Category	Application	References
Computer Vision (CV)	Object Detection	[234, 235, 236, 237, 238, 239]
	Semantic Segmentation	[240, 126, 241, 242, 243, 244, 245]
	Person Re-identification	[246]
	Super-Resolution	[247, 248, 249]
	Image Restoration	[250]
	Generative Adversarial Network (GAN)	[251, 252, 253, 254]
	Graph Neural Network (GNN)	[255]
	Disparity Estimation	[256]
	Video task	[257, 258, 259, 260]
Natural Language Processing (NLP)	Translation	[261]
	Language Modeling	[262]
	Entity Recognition	[262]
	Text Classification	[263]
	Sequential Labeling	[263]
	Keyword Spotting	[264]
Others	Network Compression	[265, 266, 267, 268, 269, 270, 271, 272]
	Federate Learning	[273, 274]
	Loss function search	[275, 276]
	Activation function search	[277]
	Image caption	[278, 279]
	Text to Speech (TTS)	[199]
	Recommendation System	[280, 281, 282]

Table 5: The summary of the existing NAS applications.

not break away from the human design paradigm. AutoML-Zero [283] 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 the evolutionary algorithm to discover complete machine learning algorithms. AutoML-Zero successfully designs two-layer neural networks based on those basic mathematical operations. Although the network searched by AutoML-Zero is much simpler compared to 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, how to design a more general, flexible, and free of human bias search space and how to discover novel neural architectures based on this search space would be challenging and advantageous.

7.2. Applying NAS to 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 manually-designed models. Additionally, many recent studies have also applied NAS to other CV tasks (shown in Table 5).

However, in terms of the NLP task, most NAS studies only implement experiments on the PTB dataset. Besides, a few NAS studies also try to apply NAS to other NLP tasks (shown in Table 5). However, as Figure 22 shows, there is still a large gap between the performance of NAS-designed models ([13, 17, 12]) and human-designed models (GPT-2 [284], FRAGE AWD-LSTM-Mos [4], adversarial AWD-LSTM-Mos [285] and Transformer-XL [5]) on the PTB dataset. Therefore, the NAS community still has a long way to go to achieve comparable results to those models designed by experts on NLP tasks.

In addition to the CV and NLP tasks, Table 5 also shows that NAS has been applied to a variety of tasks, such as network compression, federate learning, image caption, recommendation system, and even searching for loss function. Therefore, these interesting work also indicates the potential for NAS to be applied in other fields.

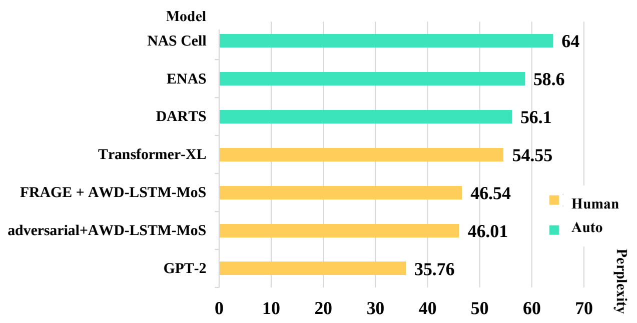


Figure 22: The SOTA models on the PTB dataset. The lower the perplexity, the better the performance. The green bar represents the automatically generated model, and the yellow bar represents the model designed by human experts. Best viewed in color.

7.3. Interpretability

Although AutoML algorithms can find promising configuration settings more efficiently than humans, there is a lack of scientific evidence to illustrate why the found settings perform better. For example, in BlockQNN [16], it's not clear 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 [230, 286, 95] The explanation for these occurrences is usually hindsight and lacks rigorous mathematical proof. Therefore, increasing the mathematical interpretability of AutoML is also an important future research direction.

7.4. Reproducibility

It is well known that 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, but the original papers do not cover too much detail. For instance, Yang et al. [120] experimentally demonstrate that the seed plays an important role in NAS experiments, while most NAS methods didn't mention the seed set by their experiments in the papers. Besides, huge resource consumption is another obstacle to reproduction. In this context, several NAS-Bench datasets are proposed, such as NAS-Bench-101 [222], NAS-Bench-201 [223], and NAS-Bench-NLP [224]. These datasets allow NAS researchers to focus on the design of optimization algorithms without wasting a lot of time on model estimation.

7.5. Robustness

NAS has been proven effective in searching for promising architectures on many open datasets (e.g., CIFAR-10 and ImageNet). Those datasets are usually for the research purpose; therefore, most of the images are well-labeled. However, in real-world situations, the data inevitably contains noise (e.g., mislabeling and inadequate information). Even worse, the data might be modified to be adversarial data with carefully designed noises. Deep learning models are easily fooled by adversarial data, and so is NAS. Currently, there are a few studies [287, 288, 289, 290] making efforts to boost the robustness of NAS against adversarial data. Guo et al. [288] experimentally explore the intrinsic impact of network architectures on network robustness against adversarial attacks, and observe that densely connected architectures tend to be more robust. They also find that the flow of solution procedure (FSP) matrix [291] is a good indicator of network robustness, i.e., the lower the FSP matrix loss more robust the network is. Chen et al. [289] propose a robust loss function to alleviate the performance degradation under symmetric label noise effectively. In [290], the authors adopt an evolutionary algorithm to search for robust architectures from a well-designed and vast search space, where many types of adversarial attacks are used as the fitness function to evaluate the robustness of neural architectures.

7.6. Joint Hyperparameter and Architecture Optimization

Most NAS studies consider hyperparameter optimization (HPO) and architecture optimization (AO) as two separate processes. However, as noted already in Section 4, there is a tremendous overlap between the methods used in HPO and AO, e.g., both of them apply random search, Bayesian optimization, and gradient-based optimization methods. In other words, it would be feasible to jointly optimize both hyperparameters and architectures. Several studies [232, 231, 233] also confirm this possibility experimentally. Thus, how to solve the problem of joint hyperparameter and architecture optimization (HAO) elegantly is a worthy studying issue.

7.7. Complete AutoML Pipeline

There are already many AutoML pipeline libraries proposed, but most of them only focus on some parts of the AutoML pipeline (Figure 1). For instance, TPOT [292], Auto-WEAK [174], and Auto-Sklearn [293] are built on top of scikit-learn [294] for building classification and regression pipelines, but they only search for the traditional ML models (like SVM and KNN). Although TPOT involves neural networks (using Pytorch [295] backend), it only supports a multi-layer perception network. Besides, Auto-Keras [22] is an open-source library developed based on Keras [296]. It focuses more on searching for deep learning models and supports multi-modal and multi-task. NNI [297] is a more powerful toolkit of AutoML, as its built-in capability contains automated feature engineering, hyper-parameter optimization, and neural architecture search. Additionally, the NAS module in NNI supports both Pytorch [295] and Tensorflow [298] and reproduces many SOTA NAS methods [13, 17, 129, 125, 194, 177, 222], which is very friendly for NAS researcher and developer. Besides, NNI also integrates the features of scikit-learn [294], which is one step closer to achieving a complete pipeline. Similarly, Vega [299] is another AutoML algorithm tool, which constructs a complete pipeline covering a set of highly decoupled functions: data augmentation, HPO, NAS, model compression, and fully train. In a word, designing an easy-to-use and complete AutoML pipeline system is a promising research direction.

7.8. Lifelong Learning

Last but not least, 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 can 1) efficiently **learn new data**; 2) meanwhile **remember old knowledge**.

7.8.1. Learn new data

Firstly, 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, but current DL models must be trained on a large number of 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, for the new task, they have to search for a new architecture; otherwise, the old architecture might not be optimal. Several studies [300, 301, 302, 303] combine meta-learning and NAS to solve this problem. A recent work by Lian et al. [302] proposes a novel and meta-learning based *transferable neural architecture search (T-NAS)* method to generate a meta-architecture, which can be adapted to new tasks easily and quickly through a few gradient steps. Another challenge of learning new data is few-shot learning scenarios, where there is only a limited number of data for the new tasks. [301] and [300] apply NAS to few-shot learning to overcome this challenge, while they only search for the most promising architecture and optimize it to work on multiple few-shot learning tasks. Elsken et al. [303] propose 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. [304] propose a general problem setup, namely *unsupervised neural architecture search (UnNAS)*, to explore whether labels are necessary for NAS. They experimentally demonstrate that the architectures searched without labels are competitive with those searched with labels; therefore, the results show that labels are not necessary for NAS and provoke people to think about what exactly affects NAS.

7.8.2. Remember old knowledge

In addition, an AutoML system must be able to constantly acquire and learn from new data, without forgetting the knowledge from old data. However, when we use new datasets to train a pretrained model, the performance of the model on the previous data sets will be greatly reduced. Incremental learning may alleviate this problem. For example, Li and Hoiem [305] propose the *learning without forgetting* (LwF) method, which trains the model using only new data, while preserving its original capabilities. In addition, iCaRL [306] makes progress based on LwF. It only uses a small part of old data for pretraining, and then gradually increases the proportion of a new class of data that is used to train the model.

8. Conclusions

In this paper, we provide a detailed and systematic review of the studies of AutoML according to the ML pipeline (see Figure 1), ranging from data preparation to model estimation. Additionally, we compare the performance and efficiency of existing NAS algorithms on the CIFAR-10 and ImageNet dataset, and provide an in-depth discussion of different research direction for NAS: one/two-stage NAS, one-shot NAS, and joint hyper-parameter and architecture optimization (HAO). We also describe several interesting and important 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.

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, 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, International Journal of Computer Vision (IJCV) 115 (3) (2015) 211–252. `doi:10.1007/s11263-015-0816-y`.

[7] K. Simonyan, A. Zisserman, Very deep convolutional networks for large-scale image recognition, arXiv preprint arXiv:1409.1556.

[8] M. Zoller, M. F. Huber, Benchmark and survey of automated machine learning frameworks., arXiv: Learning.

[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, arXiv preprint arXiv:1810.13306.

[10] T. Elsken, J. H. Metzen, F. Hutter, Neural architecture search: A survey.

[11] C. Sciuto, K. Yu, M. Jaggi, C. Musat, M. Salzmann, Evaluating the search phase of neural architecture search, arXiv preprint arXiv:1902.08142.

[12] B. Zoph, Q. V. Le, Neural architecture search with reinforcement learning, 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, arXiv preprint arXiv:1802.03268.

[14] A. Brock, T. Lim, J. M. Ritchie, N. Weston, Smash: one-shot model architecture search through hypernetworks, 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 blockwise neural network architecture generation (2018) 2423–2432.

[17] H. Liu, K. Simonyan, Y. Yang, Darts: Differentiable architecture search, arXiv preprint arXiv:1806.09055.

[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 (2018) 19–34.

[19] H. Liu, K. Simonyan, O. Vinyals, C. Fernando, K. Kavukcuoglu, Hierarchical representations for efficient architecture search, 2017.

[20] T. Chen, I. Goodfellow, J. Shlens, Net2net: Accelerating learning via knowledge transfer, 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, arXiv preprint arXiv:1611.02167.

[24] K. O. Stanley, R. Miikkulainen, Evolving neural networks through augmenting topologies, Evolutionary computation 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 (2017) 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, arXiv preprint arXiv:1804.09081.

[28] M. Suganuma, S. Shirakawa, T. Nagao, A genetic programming approach to designing convolutional neural network architectures (2017) 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 (2019) 293–312.

[30] L. Xie, A. Yuille, Genetic cnn (2017) 1379–1388.

[31] K. Ahmed, L. Torresani, Maskconnect: Connectivity learning by gradient descent (2018) 349–365.

[32] R. Shin, C. Packer, D. Song, Differentiable neural network architecture search.

[33] H. Mendoza, A. Klein, M. Feurer, J. T. Springenberg, F. Hutter, Towards automatically-tuned neural networks (2016) 58–65.

[34] A. Zela, A. Klein, S. Falkner, F. Hutter, Towards automated deep learning: Efficient joint neural architecture and hyperparameter search, 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, arXiv preprint arXiv:1605.07079.

[36] S. Falkner, A. Klein, F. Hutter, Practical hyperparameter optimization for deep learning.

[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, 2011, pp. 507–523.

[38] S. Falkner, A. Klein, F. Hutter, Bohb: Robust and efficient hyperparameter optimization at scale, 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.

[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`arXiv:1811.09828`.

[42] Y. Chen, G. Meng, Q. Zhang, S. Xiang, C. Huang, L. Mu, X. Wang, Reinforced evolutionary neural architecture search, 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 Transactions on Evolutionary Computation.

[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, 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, arXiv preprint arXiv:1906.02287.

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

[49] A. Krizhevsky, V. Nair, G. Hinton, The cifar-10 dataset, 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 Transactions on Image Processing 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, Computer Vision and Image Understanding 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 Transactions on Knowledge and Data Engineering.

[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, Journal of Computer Science and Technology 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, Journal of artificial intelligence research 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 Explorations Newsletter 6 (1) (2004) 30–39.

[68] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, W. Zaremba, Openai gym, 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, arXiv preprint arXiv:1912.09678.

[70] N. Ruiz, S. Schulter, M. Chandraker, Learning to simulate, arXiv preprint arXiv:1810.02513.

[71] T. Karras, S. Laine, T. Aila, A style-based generator architecture for generative adversarial networks, arXiv preprint arXiv:1812.04948.

[72] 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.

[73] 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.

[74] L. Sixt, Rendergan: Generating realistic labeled data–with an application on decoding bee tags, unpublished Bachelor Thesis, Freie Universität, Berlin.

[75] 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, arXiv preprint arXiv:1810.10863.

[76] N. Park, M. Mohammadi, K. Gorde, S. Jajodia, H. Park, Y. Kim, Data synthesis based on generative adversarial networks, Proceedings of the VLDB Endowment 11 (10) (2018) 1071–1083.

[77] L. Xu, K. Veeramachaneni, Synthesizing tabular data using generative adversarial networks, arXiv preprint arXiv:1811.11264.

[78] D. Donahue, A. Rumshisky, Adversarial text generation without reinforcement learning, arXiv preprint arXiv:1810.06640.

[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, Indonesian Journal of Electrical Engineering and Computer Science 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, arXiv preprint arXiv:1711.01299.

[85] S. Krishnan, E. Wu, Alphaclean: Automatic generation of data cleaning pipelines, arXiv preprint arXiv:1904.11827.

[86] I. F. Ilyas, Effective data cleaning with continuous evaluation., IEEE Data Eng. Bull. 39 (2) (2016) 38–46.

[87] M. Mahdavi, F. Neutatz, L. Visengeriyeva, Z. Abedjan, Towards automated data cleaning workflows, Machine Learning 15 (2019) 16.

[88] T. DeVries, G. W. Taylor, Improved regularization of convolutional neural networks with cutout, arXiv preprint arXiv:1708.04552.

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

[90] 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, `https://github.com/aleju/imgaug`, online; accessed 01-Feb-2020 (2020).

[91] E. K. V. I. I. A. Buslaev, A. Parinov, A. A. Kalinin, Albumentations: fast and flexible image augmentations, ArXiv e-prints`arXiv:1809.06839`.

[92] 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.

[93] 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.

[94] A. Antoniou, A. Storkey, H. Edwards, Data augmentation generative adversarial networks, arXiv preprint arXiv:1711.04340.

[95] S. C. Wong, A. Gatt, V. Stamatescu, M. D. McDonnell, Understanding data augmentation for classification: when to warp?, arXiv preprint arXiv:1609.08764.

[96] 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, arXiv preprint arXiv:1703.02573.

[97] 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, arXiv preprint arXiv:1804.09541.

[98] E. Ma, Nlp augmentation, `https://github.com/makcedward/nlpaug` (2019).

[99] 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.

[100] Y. Li, G. Hu, Y. Wang, T. Hospedales, N. M. Robertson, Y. Yang, Dada: Differentiable automatic data augmentation, arXiv preprint arXiv:2003.03780.

[101] R. Hataya, J. Zdenek, K. Yoshizoe, H. Nakayama, Faster autoaugment: Learning augmentation strategies using backpropagation, arXiv preprint arXiv:1911.06987.

[102] S. Lim, I. Kim, T. Kim, C. Kim, S. Kim, Fast autoaugment, in: Advances in Neural Information Processing Systems, 2019, pp. 6662–6672.

[103] A. Naghizadeh, M. Abavisani, D. N. Metaxas, Greedy autoaugment, arXiv preprint arXiv:1908.00704.

[104] D. Ho, E. Liang, I. Stoica, P. Abbeel, X. Chen, Population based augmentation: Efficient learning of augmentation policy

schedules, arXiv preprint arXiv:1905.05393.

[105] T. Niu, M. Bansal, Automatically learning data augmentation policies for dialogue tasks, arXiv preprint arXiv:1909.12868.

[106] M. Geng, K. Xu, B. Ding, H. Wang, L. Zhang, Learning data augmentation policies using augmented random search, arXiv preprint arXiv:1811.04768.

[107] X. Zhang, Q. Wang, J. Zhang, Z. Zhong, Adversarial autoaugment, arXiv preprint arXiv:1912.11188.

[108] 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.

[109] T. C. LingChen, A. Khonsari, A. Lashkari, M. R. Nazari, J. S. Sambee, M. A. Nascimento, Uniformaugment: A search-free probabilistic data augmentation approach, arXiv preprint arXiv:2003.14348.

[110] H. Motoda, H. Liu, Feature selection, extraction and construction, Communication of IICM (Institute of Information and Computing Machinery, Taiwan) Vol 5 (67-72) (2002) 2.

[111] M. Dash, H. Liu, Feature selection for classification, Intelligent data analysis 1 (1-4) (1997) 131–156.

[112] M. J. Pazzani, Constructive induction of cartesian product attributes, in: Feature Extraction, Construction and Selection, Springer, 1998, pp. 341–354.

[113] 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.

[114] J. Gama, Functional trees, Machine Learning 55 (3) (2004) 219–250.

[115] H. Vafaie, K. De Jong, Evolutionary feature space transformation, in: Feature Extraction, Construction and Selection, Springer, 1998, pp. 307–323.

[116] P. Sondhi, Feature construction methods: a survey, sifaka. cs. uiuc. edu 69 (2009) 70–71.

[117] 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.

[118] 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.

[119] O. Irsoy, E. Alpaydın, Unsupervised feature extraction with autoencoder trees, Neurocomputing 258 (2017) 63–73.

[120] A. Yang, P. M. Esperança, F. M. Carlucci, Nas evaluation is frustratingly hard, arXiv preprint arXiv:1912.12522.

[121] 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.

[122] F. Yu, V. Koltun, Multi-scale context aggregation by dilated convolutions, arXiv preprint arXiv:1511.07122.

[123] 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.

[124] 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.

[125] 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.

[126] 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 (2019) 82–92.

[127] 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.

[128] 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.

[129] H. Cai, L. Zhu, S. Han, Proxylessnas: Direct neural architecture search on target task and hardware, arXiv preprint arXiv:1812.00332.

[130] 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.

[131] G. Hinton, O. Vinyals, J. Dean, Distilling the knowledge in a neural network, arXiv preprint arXiv:1503.02531.

[132] 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.

[133] T. Wei, C. Wang, C. W. Chen, Modularized morphing of neural networks, arXiv preprint arXiv:1701.03281.

[134] 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.

[135] 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.

[136] H. Cai, J. Yang, W. Zhang, S. Han, Y. Yu, Path-level network transformation for efficient architecture search, arXiv preprint arXiv:1806.02639.

[137] J. Fang, Y. Sun, K. Peng, Q. Zhang, Y. Li, W. Liu, X. Wang, Fast neural network adaptation via parameter remapping and architecture search, arXiv preprint arXiv:2001.02525.

[138] 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.

[139] M. Tan, Q. V. Le, Efficientnet: Rethinking model scaling for convolutional neural networks, arXiv preprint arXiv:1905.11946.

[140] 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.

[141] J. F. Miller, S. L. Smith, Redundancy and computational efficiency in cartesian genetic programming, IEEE Transactions on Evolutionary Computation 10 (2) (2006) 167–174.

[142] F. Gruau, Cellular encoding as a graph grammar, in: IEEE Colloquium on Grammatical Inference: Theory, Applications & Alternatives, 1993.

[143] 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.

[144] M. Kim, L. Rigazio, Deep clustered convolutional kernels, in: Feature Extraction: Modern Questions and Challenges, 2015, pp. 160–172.

[145] 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.

[146] H. Zhu, Z. An, C. Yang, K. Xu, E. Zhao, Y. Xu, Eena: Efficient evolution of neural architecture (2019). `arXiv:1905.07320`.

[147] R. J. Williams, Simple statistical gradient-following algorithms for connectionist reinforcement learning, Machine learning 8 (3-4) (1992) 229–256.

[148] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, O. Klimov, Proximal policy optimization algorithms, arXiv preprint arXiv:1707.06347.

[149] 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.

[150] C. He, H. Ye, L. Shen, T. Zhang, Milenas: Efficient neural architecture search via mixed-level reformulation (2020). arXiv:2003.12238.

[151] X. Dong, Y. Yang, Searching for a robust neural architecture in four gpu hours (2019). arXiv:1910.04465.

[152] S. Xie, H. Zheng, C. Liu, L. Lin, Snas: stochastic neural architecture search, arXiv preprint arXiv:1812.09926.

[153] 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.

[154] E. Jang, S. Gu, B. Poole, Categorical reparameterization with gumbel-softmax, arXiv preprint arXiv:1611.01144.

[155] C. J. Maddison, A. Mnih, Y. W. Teh, The concrete distribution: A continuous relaxation of discrete random variables, arXiv preprint arXiv:1611.00712.

[156] H. Liang, S. Zhang, J. Sun, X. He, W. Huang, K. Zhuang, Z. Li, Darts+: Improved differentiable architecture search with early stopping, arXiv preprint arXiv:1909.06035.

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

[158] R. Negrinho, G. Gordon, Deeparchitect: Automatically designing and training deep architectures (2017). arXiv:1704.08792.

[159] R. Negrinho, D. Patil, N. Le, D. Ferreira, M. Gormley, G. Gordon, Towards modular and programmable architecture search (2019). arXiv:1909.13404.

[160] G. Dikov, P. van der Smagt, J. Bayer, Bayesian learning of neural network architectures (2019). arXiv:1901.04436.

[161] C. White, W. Neiswanger, Y. Savani, Bananas: Bayesian optimization with neural architectures for neural architecture search (2019). arXiv:1910.11858.

[162] 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.

[163] J.-M. Perez-Rua, M. Baccouche, S. Pateux, Efficient progressive neural architecture search (2018). arXiv:1808.00391.

[164] C. E. Rasmussen, Gaussian processes in machine learning, Lecture Notes in Computer Science (2003) 63–71.

[165] 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.

[166] 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.

[167] M. M. Ian Dewancker, S. Clark, Bayesian optimization primer. URL https://app.sigopt.com/static/pdf/SigOpt_Bayesian_Optimization_Primer.pdf

[168] B. Shahriari, K. Swersky, Z. Wang, R. P. Adams, N. De Freitas, Taking the human out of the loop: A review of bayesian optimization, Proceedings of the IEEE 104 (1) (2016) 148–175.

[169] 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.

[170] 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.

[171] J. Stork, M. Zaefferer, T. Bartz-Beielstein, Improving neuroevolution efficiency by surrogate model-based optimization with phenotypic distance kernels (2019). arXiv:1902.03419.

[172] 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.

[173] A. Camero, H. Wang, E. Alba, T. Bäck, Bayesian neural architecture search using a training-free performance metric (2020). arXiv:2001.10726.

[174] C. Thornton, F. Hutter, H. H. Hoos, K. Leyton-Brown, Autoweka: 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.

[175] A. sharpdarts, V. Jain, G. D. Hager, sharpdarts: Faster and more accurate differentiable architecture search, Tech. rep. (2019).

[176] Y. Geifman, R. El-Yaniv, Deep active learning with a neural architecture search, in: Advances in Neural Information Processing Systems, 2019, pp. 5974–5984.

[177] L. Li, A. Talwalkar, Random search and reproducibility for neural architecture search, arXiv preprint arXiv:1902.07638.

[178] J. Bergstra, Y. Bengio, Random search for hyper-parameter optimization, Journal of machine learning research 13 (Feb) (2012) 281–305.

[179] C.-W. Hsu, C.-C. Chang, C.-J. Lin, et al., A practical guide to support vector classification.

[180] J. Y. Hesterman, L. Caucci, M. A. Kupinski, H. H. Barrett, L. R. Furenlid, Maximum-likelihood estimation with a contracting-grid search algorithm, IEEE transactions on nuclear science 57 (3) (2010) 1077–1084.

[181] L. Li, K. Jamieson, G. DeSalvo, A. Rostamizadeh, A. Talwalkar, Hyperband: A novel bandit-based approach to hyperparameter optimization, The Journal of Machine Learning Research 18 (1) (2017) 6765–6816.

[182] M. Feurer, F. Hutter, Hyperparameter Optimization, Springer International Publishing, Cham, 2019, pp. 3–33. URL https://doi.org/10.1007/978-3-030-05318-5_1

[183] T. Yu, H. Zhu, Hyper-parameter optimization: A review of algorithms and applications, arXiv preprint arXiv:2003.05689.

[184] Y. Bengio, Gradient-based optimization of hyperparameters, Neural computation 12 (8) (2000) 1889–1900.

[185] J. Domke, Generic methods for optimization-based modeling, in: Artificial Intelligence and Statistics, 2012, pp. 318–326.

[186] D. Maclaurin, D. Duvenaud, R. Adams, Gradient-based hyperparameter optimization through reversible learning, in: International Conference on Machine Learning, 2015, pp. 2113–2122.

[187] F. Pedregosa, Hyperparameter optimization with approximate gradient, arXiv preprint arXiv:1602.02355.

[188] L. Franceschi, M. Donini, P. Frasconi, M. Pontil, Forward and reverse gradient-based hyperparameter optimization, arXiv preprint arXiv:1703.01785.

[189] 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, arXiv preprint arXiv:1909.13371.

[190] D. P. Kingma, J. Ba, Adam: A method for stochastic optimization, arXiv preprint arXiv:1412.6980.

[191] P. Chrabaszcz, I. Loshchilov, F. Hutter, A downsampled variant of imagenet as an alternative to the CIFAR datasets, CoRR abs/1707.08819. arXiv:1707.08819. URL http://arxiv.org/abs/1707.08819

[192] 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).

[193] C. Wong, N. Houlsby, Y. Lu, A. Gesmundo, Transfer learning with neural automl, in: Advances in Neural Information Processing Systems, 2018, pp. 8356–8365.

[194] 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, arXiv preprint arXiv:1904.02877.

[195] K. Eggensperger, F. Hutter, H. H. Hoos, K. Leyton-Brown, Surrogate benchmarks for hyperparameter optimization., in: MetaSel@ ECAI, 2014, pp. 24–31.

[196] C. Wang, Q. Duan, W. Gong, A. Ye, Z. Di, C. Miao, An evaluation of adaptive surrogate modeling based optimization with two benchmark problems, Environmental Modelling & Software 60 (2014) 167–179.

[197] K. Eggensperger, F. Hutter, H. Hoos, K. Leyton-Brown, Ef-

ficient benchmarking of hyperparameter optimizers via surrogates, in: Twenty-Ninth AAAI Conference on Artificial Intelligence, 2015.

[198] K. K. Vu, C. D'Ambrosio, Y. Hamadi, L. Liberti, Surrogate-based methods for black-box optimization, International Transactions in Operational Research 24 (3) (2017) 393–424.

[199] R. Luo, X. Tan, R. Wang, T. Qin, E. Chen, T.-Y. Liu, Semi-supervised neural architecture search (2020). `arXiv:2002.10389`.

[200] A. Klein, S. Falkner, J. T. Springenberg, F. Hutter, Learning curve prediction with bayesian neural networks.

[201] B. Deng, J. Yan, D. Lin, Peephole: Predicting network performance before training, arXiv preprint arXiv:1712.03351.

[202] 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.

[203] M. Mahsereci, L. Balles, C. Lassner, P. Hennig, Early stopping without a validation set, arXiv preprint arXiv:1703.09580.

[204] 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, arXiv preprint arXiv:1806.10332.

[205] D. Han, J. Kim, J. Kim, Deep pyramidal residual networks (2016). `arXiv:1610.02915`.

[206] 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.

[207] 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.

[208] H. Zhou, M. Yang, J. Wang, W. Pan, Bayesnas: A bayesian approach for neural architecture search (2019). `arXiv:1905.04919`.

[209] 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`.

[210] 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.

[211] 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.

[212] C. Zhang, M. Ren, R. Urtasun, Graph hypernetworks for neural architecture search (2018). `arXiv:1810.05749`.

[213] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, L.-C. Chen, Mobilenetv2: Inverted residuals and linear bottlenecks (2018). `arXiv:1801.04381`.

[214] 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`.

[215] H. Cai, C. Gan, S. Han, Once for all: Train one network and specialize it for efficient deployment, arXiv preprint arXiv:1908.09791.

[216] J. Mei, Y. Li, X. Lian, X. Jin, L. Yang, A. Yuille, J. Yang, Atomnas: Fine-grained end-to-end neural architecture search, arXiv preprint arXiv:1912.09640.

[217] 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`.

[218] 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.

[219] 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.

[220] 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`.

[221] M. G. Kendall, A new measure of rank correlation, Biometrika 30 (1/2) (1938) 81–93.
URL `http://www.jstor.org/stable/2332226`

[222] C. Ying, A. Klein, E. Real, E. Christiansen, K. Murphy, F. Hutter, NAS-Bench-101: Towards Reproducible Neural Architecture Search, arXiv e-prints.

[223] X. Dong, Y. Yang, Nas-bench-201: Extending the scope of reproducible neural architecture search, in: International Conference on Learning Representations, 2020.
URL `https://openreview.net/forum?id=HJxyZkBKDr`

[224] 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`.

[225] X. Zhang, Z. Huang, N. Wang, You only search once: Single shot neural architecture search via direct sparse optimization, arXiv preprint arXiv:1811.01567.

[226] 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, arXiv preprint arXiv:2003.11142.

[227] X. Chu, B. Zhang, R. Xu, J. Li, Fairnas: Rethinking evaluation fairness of weight sharing neural architecture search, arXiv preprint arXiv:1907.01845.

[228] Y. Benyahia, K. Yu, K. Bennani-Smires, M. Jaggi, A. Davison, M. Salzmann, C. Musat, Overcoming multi-model forgetting (2019). `arXiv:1902.08232`.

[229] S. P. X. C. S. S. Miao Zhang, Huiqi Li, 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.

[230] G. Bender, Understanding and simplifying one-shot architecture search.

[231] 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`.

[232] A. Klein, F. Hutter, Tabular benchmarks for joint architecture and hyperparameter optimization, arXiv preprint arXiv:1905.04970.

[233] 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, arXiv preprint arXiv:2006.02049.

[234] G. Ghiasi, T.-Y. Lin, Q. V. Le, Nas-fpn: Learning scalable feature pyramid architecture for object detection (2019) 7036–7045.

[235] 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.

[236] M. Tan, R. Pang, Q. V. Le, Efficientdet: Scalable and efficient object detection, arXiv preprint arXiv:1911.09070.

[237] Y. Chen, T. Yang, X. Zhang, G. Meng, C. Pan, J. Sun, Detnas: Neural architecture search on object detection, arXiv preprint arXiv:1903.10979 1 (2) (2019) 4–1.

[238] J. Guo, K. Han, Y. Wang, C. Zhang, Z. Yang, H. Wu, X. Chen, C. Xu, Hit-detector: Hierarchical trinity architecture search for object detection, in: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2020, pp. 11405–11414.

[239] 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.

[240] Y. Weng, T. Zhou, Y. Li, X. Qiu, Nas-unet: Neural architecture search for medical image segmentation, IEEE Access 7 (2019)

44247–44257.

[241] 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.

[242] 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.

[243] 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.

[244] 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.

[245] 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.

[246] 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.

[247] 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.

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

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

[250] H. Zhang, Y. Li, H. Chen, C. Shen, Ir-nas: Neural architecture search for image restoration, arXiv preprint arXiv:1909.08228.

[251] 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.

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

[253] 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.

[254] 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.

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

[256] 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.

[257] 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.

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

[259] 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.

[260] 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.

[261] Y. Fan, F. Tian, Y. Xia, T. Qin, X.-Y. Li, T.-Y. Liu, Searching better architectures for neural machine translation, IEEE/ACM Transactions on Audio, Speech, and Language Processing.

[262] 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.

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

[264] 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.

[265] 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.

[266] 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.

[267] 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, pp. 0–0.

[268] 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.

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

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

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

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

[273] C. He, M. Annavaram, S. Avestimehr, Fednas: Federated deep learning via neural architecture search (2020). `arXiv:2004.08546`.

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

[275] 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.

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

[277] P. Ramachandran, B. Zoph, Q. V. Le, Searching for activation functions (2017). `arXiv:1710.05941`.

[278] H. Wang, H. Wang, K. Xu, Evolutionary recurrent neural network for image captioning, Neurocomputing.

[279] L. Wang, Y. Zhao, Y. Jinnai, Y. Tian, R. Fonseca, Neural architecture search using deep neural networks and monte carlo tree search, arXiv preprint arXiv:1805.07440.

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

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

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

[283] E. Real, C. Liang, D. R. So, Q. V. Le, Automl-zero: Evolving machine learning algorithms from scratch (2020). `arXiv:2003.03384`.

[284] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever, Language models are unsupervised multitask learners, OpenAI Blog 1 (2019) 8.

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

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

[287] S. KOTYAN, D. V. VARGAS, Is neural architecture search a way forward to develop robust neural networks?, Proceedings of the Annual Conference of JSAI JSAI2020 (2020) 2K1ES203–2K1ES203.

[288] 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.

[289] 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.

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

[291] 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.

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

[293] 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.

[294] 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, Journal of Machine Learning Research 12 (2011) 2825–2830.

[295] 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 (2019).

[296] F. Chollet, et al., Keras, `https://github.com/fchollet/keras` (2015).

[297] NNI (Neural Network Intelligence), 2020.
URL `https://github.com/microsoft/nni`

[298] 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`.

[299] Vega, 2020.
URL `https://github.com/huawei-noah/vega`

[300] R. Pasunuru, M. Bansal, Continual and multi-task architecture search, arXiv preprint arXiv:1906.05226.

[301] 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, arXiv preprint arXiv:1806.06927.

[302] 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.

[303] 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.

[304] C. Liu, P. Dollár, K. He, R. Girshick, A. Yuille, S. Xie, Are labels necessary for neural architecture search? (2020). `arXiv:2003.12056`.

[305] Z. Li, D. Hoiem, Learning without forgetting, IEEE transactions on pattern analysis and machine intelligence 40 (12) (2018) 2935–2947.

[306] 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.