Automating the Future: A Comprehensive Review of Machine Learning Pipeline Generation

Yutao Chen
School of Informatics
Universitat Politècnica de Catalunya
yutao.chen@estudiantat.upc.edu

Hieu Nguyen Minh
School of Informatics
Universitat Politècnica de Catalunya
hieu.nguyen.minh@estudiantat.upc.edu

***Abstract*—The rapid advancements in machine learning have resulted in complex models. However, the construction and optimization of ML pipelines remain labor-intensive and require substantial expertise. In this paper, we explore the state-of-the-art techniques for automating ML pipeline generation. We cover a range of methodologies including data preparation, feature engineering, hyperparameter optimization, and model selection. By examining recent advancements and evaluating various approaches, we provide a comprehensive understanding of current automated ML pipeline generation.**

***Index Terms*—automl, hyperparameter optimization**

I. Introduction

Despite the advancements in Machine Learning (ML), the creation and fine-tuning of ML pipelines remain challenging, requiring domain expertise. The ML pipeline encompasses a series of steps, including data collection, preprocessing, feature engineering, model training, and evaluation, each of which must be carefully designed and processed to achieve optimal performance. Automating these processes can significantly reduce the time and expertise needed, leading to the concept of automated ML.

Automated Machine Learning (AutoML) can be defined as an automated construction of ML pipelines that reduce the demand for data scientists by enabling domain experts on ML applications without extensive knowledge of statistics and ML [1]. As illustrated in Figure 1, an AutoML pipeline consists of four main steps: (1) data preparation, (2) feature engineering, (3) model optimization, and (4) model estimation [2]. In the first phase, we prepare a clean dataset from a raw one, probably adding augmented data, then meaning features can be extracted from the dataset in the second phase. Will all these extracted features and associated labels as the training data, we aim to create an optimized model in the third step. The model optimization consists of a *search space* and *optimization methods*. The search space classifies the models in two categories: traditional models and deep neural networks; meanwhile optimization methods contains *HyPerparameter Optimization (HPO)* and *Architecture Optimization (AO)*. The former is related to optimizing training parameters (e.g. learning rate, regularization), and the latter indicates the model architecture parameter (e.g. number of network layers, number of filters). Finally, the search space of neural networks, the AO, along with the model estimation methods, form the *Neural Architecture Search (NAS)*. NAS in [3] is implemented as a Recurrent Neural network (RNN) to generate model descriptions of neural networks and trains this RNN with reinforcement learning to maximize the expected accuracy of the generated architectures. NAS can design a novel network architecture that obtains comparable results to the human-invented one.

II. Problem Formulation

A. ML Pipeline

The ML pipeline structure can be modeled as a Directed Acyclic Graph (DAG), where nodes represent algorithms, and edges represent the flow of the dataset through different algorithms [1]. Let G be the set of valid pipeline structures and g is the number of nodes in G (i.e. the pipeline length). A ML pipline $h : \mathbb{X} \to \mathbb{Y}$ transforms a feature vector $\vec{x} \in \mathbb{X}$ into a target $\vec{y} \in \mathbb{Y}$. Let $\mathcal{A} = \left\{A^{(1)}, \ldots, A^{(R)}\right\}$ be a set of algorithms, and the hyperparameters of each algorithm $A^{(i)}$ have domain $\Lambda^{(i)}$. Then an ML pipeline is defined by a triplet $\left(g, \vec{A}, \vec{\lambda}\right)$, where $g \in G$ is a valid pipeline structure, $\vec{A} \in \mathcal{A}^{|g|}$ is a vector of selected algorithms for each node, and $\vec{\lambda}$ is a vector of hyperparameters of all algorithms. The pipeline is denoted as $\mathcal{P}_{g,\vec{A},\vec{\lambda}}$.

B. True Pipeline Performance

Let $P(\mathbb{X}, \mathbb{Y})$ be a joint probability distribution of the feature space $\mathbb{X}$ and target space $\mathbb{Y}$, then a pipeline trained on P is denoted as $\mathcal{P}_{g,\vec{A},\vec{\lambda},P}$. Given a pipeline $\mathcal{P}_{g,\vec{A},\vec{\lambda}}$, a loss function $\mathcal{L}(\cdot,\cdot)$ and a generative model $P(\mathbb{X}, \mathbb{Y})$, the performance of $\mathcal{P}_{g,\vec{A},\vec{\lambda},P}$ is calculated as:

$$R\left(\mathcal{P}_{g,\vec{A},\vec{\lambda},P}, P\right) = \mathbb{E}(\mathcal{L}(h(\mathbb{X}), \mathbb{Y})) = \int \mathcal{L}(h(\mathbb{X}), \mathbb{Y}) \mathrm{d}P(\mathbb{X}, \mathbb{Y}), \tag{1}$$

where $h(\mathbb{X})$ is the predicted output of $\mathcal{P}_{g,\vec{A},\vec{\lambda},P}$.

C. Pipeline Creation Problem

An ML task is defined by a generative model, loss function and an ML problem type [1]. Generating an ML pipeline for a given ML task consists of three sub-tasks: (1) determining pipeline structure, e.g., number of feature engineering steps and models; (2) for each step an algorithm is selected; and (3) for each selected algorithm its corresponding hyperparameters

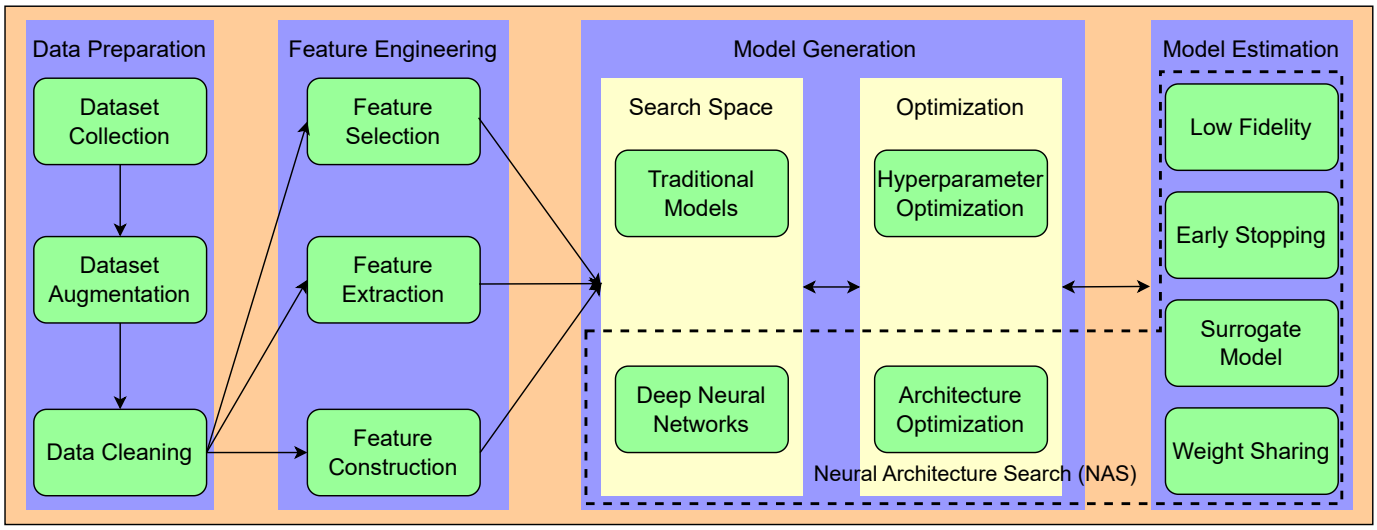

Fig. 1. AutoML pipeline (reproduced from [2]).

are selected. The pipeline creation problem iss to find a pipeline structure with a joint algorithm and hyperparameter selection that minimizes:

$$(g, \vec{A}, \vec{\lambda})^\star \in \underset{g\in G, \vec{A}\in\mathcal{A}^{|g|}, \vec{\lambda}\in\Lambda}{\arg\min} R\left(\mathcal{P}_{g,\vec{A},\vec{\lambda},P}, P\right). \quad (2)$$

Let $D_{train} = \{(x_1, y_1), \ldots, (x_n, y_n)\}$ be a training set which is split into K cross-validation folds $\left\{D_{valid}^{(1)}, \ldots, D_{valid}^{(K)}\right\}$ and $\left\{D_{train}^{(1)}, \ldots, D_{train}^{(K)}\right\}$, where $D_{train}^{(i)} = D_{train} \backslash D_{valid}^{(i)}$ for $i = 1, \ldots, K$. We can rewrite the problem as follows:

$$(g, \vec{A}, \vec{\lambda})^\star \in \underset{g\in G, \vec{A}\in\mathcal{A}^{|g|}|\vec{\lambda}\in\Lambda}{\arg\min} \frac{1}{k}\sum_{i=1}^{k} \hat{R}\left(\mathcal{P}_{g,\vec{A},\vec{\lambda},D_{train}^{(i)}}, D_{valid}^{(i)}\right). \quad (3)$$

The state-of-the-art algorithms that solve the pipeline creation problem are categorized as shown in Figure 2. NAS is an instance of structure search problem, while *Combined Algorithm Selection and Hyperparameter* (CASH) problem [4] is the following joint optimization problem. Let $\mathcal{L}\left(A_{\boldsymbol{\lambda}}^{(j)}, D_{train}^{(i)}, D_{valid}^{(i)}\right)$ denote the loss that algorithm $A^{(j)}$ achieves on $D_{valid}^{(i)}$ when trained on $D_{train}^{(i)}$ with hyperparameters $\boldsymbol{\lambda}$. The purpose of CASH problem is to find the joint algorithm and hyperparameter setting that minimizes that loss:

$$A^\star, \boldsymbol{\lambda}_\star \in \underset{A^{(j)}\in\mathcal{A}, \boldsymbol{\lambda}\in\Lambda^{(j)}}{\operatorname{argmin}} \frac{1}{K}\sum_{i=1}^{K} \mathcal{L}\left(A_{\boldsymbol{\lambda}}^{(j)}, D_{train}^{(i)}, D_{valid}^{(i)}\right). \quad (4)$$

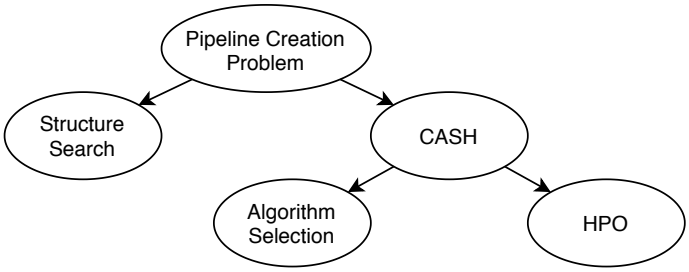

Fig. 2. Subproblems of the pipeline creation problem.

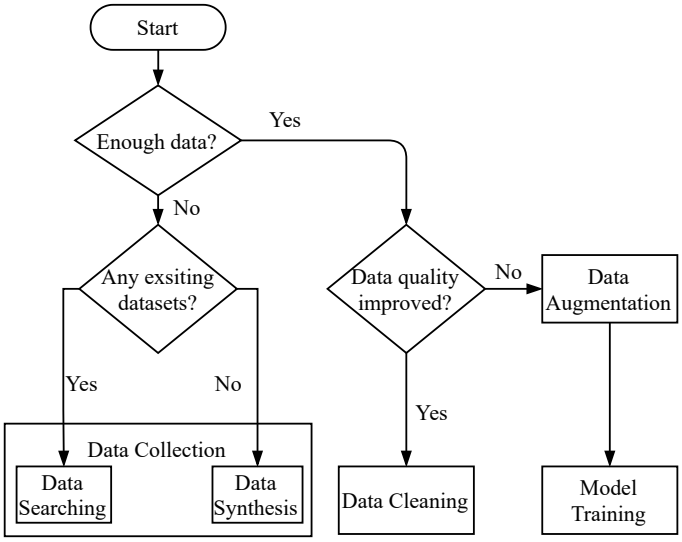

Fig. 3. Data preparation flow chart [2].

III. Data Preparation

The first step in data preparation is to obtain a raw dataset, which can be a provided one or an extended one with data augmentation. After a dataset has been obtained, data cleaning is used to remove noisy data and add missing data, so that the downstream model can be trained appropriately. A dataset can further be improved with augmented data which can potentially enhance model performance and robustness. Figure 3 describes the flow chart of data preparation.

A. Data Collection

The advancement of ML requires high-quality datasets, leading to the development numerous open datasets, such as the handwritten digit dataset MNIST[1], CIFAR-10 ann CIFAR-100[2], and ImageNet[3]. Various datasets can also be accessed through platforms such as Kaggle[4] or Google Dataset Search[5]. However, finding suitable datasets for specific tasks, especially those related to medical care or privacy, remains challenging. This issue can be solve with data searching and data synthesis. Data searching leverages the Internet resources to gather datasets, but this method faces issues like irrelevant results and incorrect or unlabeled data. Data synthesis involves generating datasets using tools like data simulators and Generative Adversarial Networks (GANs) [5]. These methods enhance the ability to create realistic and diverse datasets, thus supporting the development of robust ML models.

B. Data Augmentation

Data augmentation can generate new data based on existing data and can also serves as a regularizer to avoid overfitting during model training. For image data, affine transformations include rotation, scaling, random cropping, and reflection; elastic transformations involve contrast shift, brightness shift, blurring, and channel shuffle; advanced transformations includes random erasing, image blending. Neural-based transformations include adversarial noise [6], neural style transfer [7], and GAN technique [8]. For audio data, various techniques can be applied, such as noise injection, time-related modifications (e.g. time shift, time stretching), pitch scaling, equalization. For textual data, data warping and synthetic oversampling [9] is used to create additional training examples. NLPAug [6] is an open-source library integrating many augmentation operations for both textual and audio data. Recent methods enhance augmentation search efficiency using strategies like gradient descent [10], Bayesian optimization [11], online hyperparameter learning [12], and random search [13].

C. Data Cleaning

Collected data often contain noise that can badly affect model training, necessitating data cleaning. Traditional data cleaning uses specialist knowledge, which is limited and expensive, while other approaches clean only a small subset of the data but still have comparable results to full dataset cleaning [14], [15]. However, these methods still require a data scientist to design the cleaning operations. BoostClean [16] aims to automate this process by treating it as a boosting problem, where each cleaning operation adds a new layer to the downstream ML model, enhancing performance through boosting and feature selection. AlphaClean [17] further automates data cleaning by transforming it into a hyperparameter optimization problem, creating a series of pipelined cleaning operations from a predefined search space. Additionally, meta-learning-based cleaning techniques [18] is used to automate data cleaning.

[1]https://www.kaggle.com/datasets/hojjatk/mnist-dataset
[2]https://www.cs.toronto.edu/ kriz/cifar.html
[3]https://www.image-net.org/
[4]https://www.kaggle.com/
[5]https://datasetsearch.research.google.com/
[6]https://github.com/makcedward/nlpaug

IV. Feature Engineering

Feature engineering focuses on optimizing the extraction of features from raw data so that models can be trained on them. There are three areas in feature engineering: feature selection, feature extraction, and feature construction. Feature extraction and construction involve feature transformation and the creation of a new set of features. The former reduces the dimensionality of features through specific mapping functions, and the latter expands the original feature space. Meanwhile, feature selection aims to minimize feature redundancy by identifying the most important features.

A. Feature Selection

Feature selection builds a feature subset from the original feature set by eliminating irrelevant or redundant features, thus simplifying the model and preventing overfitting. The selected features are typically divergent and highly correlated with target values. The search strategy for feature selection involves three types of algorithms: complete search, heuristic search, and random search. Complete search includes exhaustive and non-exhaustive methods; Heuristic search includes sequential forward selection (SFS), sequential backward selection (SBS), and bidirectional search (BS); and random search methods includes simulated annealing (SA) and genetic algorithms (GAs).

B. Feature Construction

Feature construction creates new features from the basic feature space or raw data to increase the representative ability of the original features. Automatic feature construction methods [19] automates the search of operation combinations, achieving results comparable to those from human experts. After constructing a new feature, feature-selection techniques evaluate its effectiveness.

C. Feature Extraction

Feature extraction is a dimensionality-reduction process that uses mapping functions to extract informative and non-redundant features based on specific metrics, altering the original features. Traditional methods for this process includes Principal Component Analysis (PCA), independent component analysis, isomap, nonlinear dimensionality reduction, and Linear Discriminant Analysis (LDA). Recently, neural networks have been used for feature extraction, such as autoencoder-based algorithms [20], [21].

V. Model Generation

A. Search Space

1) Traditional Models: Configurations for a model consist of choices of classifiers and their hyperparameters [4], [22], forming the search space. The selection of classifiers can

be modeled as a discrete variable, where 1 indicates the classifier is used and 0 indicates it is not used. Traditional ML models includes Support Vector Machine (SVM) [23], k-nearest neighbors algorithm (KNN) [24] or Perceptron Learning Algorithm [25]. The nature of hyperparameters depends on the model's design and implementation. For example, the number of nearest neighbors is a discrete parameter, while the penalty parameter for logistic regression is continuous.

2) Deep Neural Networks:

Entire-structured Search Space: Efficient Neural Architecture Search (ENAS) [26] introduces a method to efficiently explore entire-structured search spaces for deep neural networks. This approach represents the search space as a large directed acyclic graph (DAG), where each subgraph corresponds to a possible neural network architecture. By sharing parameters among these subgraphs, ENAS reduces the computational cost and time required for neural architecture search.

Example Analysis:

The provided diagram in Figure 4 shows a convolutional neural network architecture discovered by ENAS. The nodes represent different convolutional operations with varying kernel sizes and types (e.g., separable convolutions, max pooling), while the edges denote the connections between these operations.

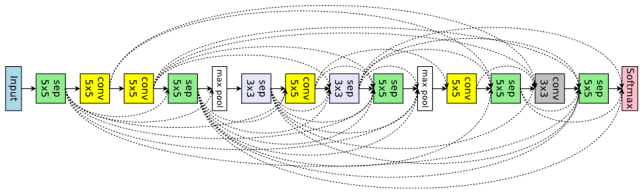

Fig. 4. Example of a convolutional neural network architecture discovered by ENAS [26]. Nodes represent different operations, and edges represent the flow of information.

This architecture can be analyzed as follows:

1) Nodes and Edges: Each node v_i performs a specific operation, such as a 5x5 separable convolution (sep 5x5) or a 3x3 convolution (conv 3x3). The edges represent the information flow between these operations.
2) Parameter Sharing: The parameters used in each operation are shared across different architectures sampled by the controller. This sharing significantly reduces the computational cost.
3) Optimization: The optimization alternates between updating the shared parameters ω to minimize the loss function and updating the controller parameters θ to maximize the reward function.
4) Evaluation: The performance of each architecture is evaluated based on its accuracy or loss on a validation set, and the best-performing architectures are selected for further training.

This approach allows ENAS to efficiently explore a vast search space of potential architectures, leading to the discovery of high-performing models with significantly reduced computational resources compared to traditional NAS methods [26].

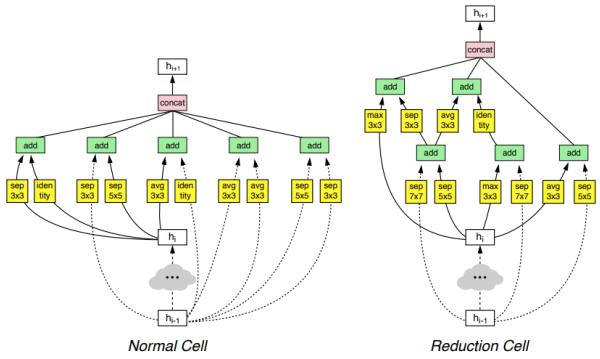

Fig. 5. Cell-based search space [27].

Cell-based search space

The cell-based search space approach involves creating a search space composed of modular units, called cells. Each cell is essentially a small, repeatable network structure. There are two main types of cells: normal cells and reduction cells. Normal cells maintain the spatial dimensions of the input, while reduction cells reduce the spatial dimensions by half [27].

As Figure 5, each cell can be represented as:

- **Normal Cell:** Maintains the spatial dimensions of the input.
- **Reduction Cell:** Reduces the spatial dimensions by a factor of 2.

For a given cell, let x_i and x_{i-1} be the inputs from the two previous layers.

$$h_i = \text{Operation}(x_i) + \text{Operation}(x_{i-1}), \tag{5}$$

where Operation can be one of the following:

- Separable convolution: sep $k \times k$
- Average pooling: avg $k \times k$
- Identity mapping

The final output of the cell is the concatenation of all h_i:

$$\text{Output} = \text{concat}(h_1, h_2, \ldots, h_n). \tag{6}$$

Hierarchical Search Space

The hierarchical search space in neural architecture search (NAS) extends beyond the cell-based approach by including the network level structure in addition to the cell level [28]. This allows for a more comprehensive exploration of architectural variations, especially important for tasks like semantic segmentation where spatial resolution changes are crucial [29].

Network Level: The network level architecture controls the spatial resolution changes throughout the network [30]. As shown in the diagram 6, each layer in the network can either maintain, halve, or double the spatial resolution. This is represented by the following update equation [29]:

$$sH_l = \beta_{s/2\to s} \cdot \text{Cell}(sH_{l-1}, sH_{l-2}; \alpha) \tag{7}$$

$$+ \beta_{s\to s} \cdot \text{Cell}(sH_{l-1}, sH_{l-2}; \alpha) \tag{8}$$

$$+ \beta_{2s\to s} \cdot \text{Cell}(2sH_{l-1}, sH_{l-2}; \alpha) \tag{9}$$

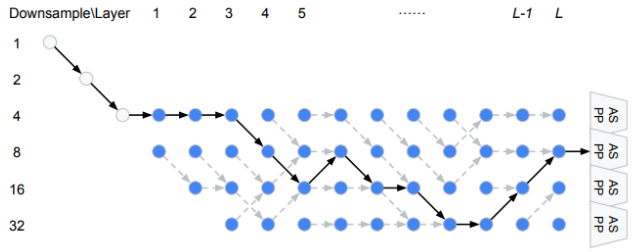

Fig. 6. Hierarchical Search Space in Deep Neural Networks [29]

where s denotes the spatial resolution, β are the learnable weights, and α are the parameters defining the cell architecture.

The hierarchical search space includes both the cell and network levels, allowing for a joint optimization of the repeatable cell structure and the overall network architecture. This approach enables the design of highly efficient and effective neural networks tailored to specific tasks like Single Image Super-Resolution [31].

Morphism-based Search Space

In deep neural networks, a morphism-based search space allows for the transformation of an existing neural network into a new one while preserving its original function [32]. This process is called **network morphism**. The goal is to enable the child network to inherit knowledge from the parent network and continue learning with a shortened training time.

Identity Morphism (IdMorph) The simplest form of network morphism is the identity morphism (IdMorph), which involves adding identity mappings between layers. For a given layer transformation represented as $B_{l+1} = G(B_l)$, where B_l and B_{l+1} are the input and output of layer G, an IdMorph transformation can be inserted without altering the network's function. Mathematically, this can be written as:

$$B_{l+1} = G(B_l) \rightarrow B_{l+1} = G_2(G_1(B_l)). \tag{10}$$

where G_1 and G_2 are transformations such that $G = G_2 \circ G_1$.

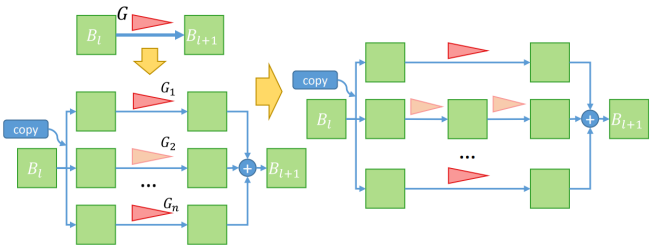

Fig. 7. Illustration of network morphism. The parent network (left) is transformed into a deeper and wider child network (right) through network morphism techniques [32].

Network Morphism While IdMorph is limited to inserting identity mappings, network morphism extends this concept by allowing for changes in depth, width, and kernel size of the network layers. Consider the original layer transformation $B_{l+1} = G(B_l)$. In the case of depth morphing, a new layer B_i is inserted such that [32]:

$$B_{l+1} = G(B_l) \rightarrow B_{l+1} = G_2(G_1(B_l)), \tag{11}$$

where G_1 and G_2' are transformations that increase the width of the layer. Consider a parent network with a single transformation $B_{l+1} = G(B_l)$. By applying network morphism, we can insert multiple transformations as follows:

$$B_{l+1} = F_3(F_2(F_1(B_l))). \tag{12}$$

Here, F_1, F_2, and F_3 are chosen such that the overall function of the network remains unchanged but allows for further learning and adaptation.

The advantages of network morphism include reduced training time and improved performance due to internal regularization [33] [34]. These benefits make network morphism a powerful tool for designing and optimizing deep neural networks.

B. Hyperparameter Optimization

1) Grid Search and Random Search: Grid search divides the search space into regular intervals and selects the best-performing point after evaluating all points, while random search selects the best point from a set of randomly drawn points. Grid search is simple and supports parallel implementation but is computationally expensive and inefficient for large hyperparameter spaces, which is alleviated by a coarse-to-fine grid search [35] and maximum likelihood estimation [36]. On the other hand, Random search has been proven to be more practical and efficient than grid search [37], but neither of these methods guarantee an optimum value. Based on available resources, we can balance between doing more searches to get better performance and limiting resource budgets.

2) Bayesian Optimization: Bayesian optimization constructs a probabilistic model mapping hyperparameters to objective metrics evaluated on a validation set, balancing exploration (evaluating many hyperparameter sets) and exploitation (focusing on promising hyperparameters). Common surrogate models for BO include GP, RF, and TPE. GP, a popular model, scales cubically with data samples, whereas RF handles large spaces better. The Bayesian Optimization-based hyperband (BOHB) [38] combinines TPE and hyperband, outperforming standard methods. FABOLAS [39] maps validation loss and training time to dataset size, training a generative model on progressively larger sub-datasets, achieving faster performance than other algorithms.

3) Gradient-based Optimization: Gradient-based optimization algorithms use gradient information to optimize hyperparameters. A reversible-dynamics memory-tape approach [40] manages thousands of hyperparameters using gradient information, while both reverse- and forward-mode methods [41] can also be used. Regular hyperparameters (e.g. learning rate, weight decay), optimizer's parameters (e.g. moment coefficients in Adam optimizer) are mutually optimized in an ultimate optimizer [42].

C. Architecture Optimization

1) Evolutionary Algorithm: The Evolutionary Algorithm (EA) is a robust global optimization technique inspired by the principles of natural evolution. Unlike traditional optimization methods, EA operates on a population of potential solutions and refines them through processes such as selection, crossover, and mutation. This approach is particularly effective for complex problems, where traditional methods may fall short.

In EA, the representation of neural network architectures can be categorized into two main types: direct encoding and indirect encoding. Direct encoding explicitly specifies the network structure. For example, a binary string can represent connections between nodes [43]. Meanwhile, indirect encoding specifies rules for generating the network structure, leading to a more compact representation [44]. An example is cellular encoding, which uses a set of rules to construct the network iteratively [45].

2) Reinforcement Learning: Reinforcement Learning (RL) has been successfully applied to Neural Architecture Search (NAS) to automate the design of neural networks [46]. In this approach, an agent, typically a Recurrent Neural Network (RNN), acts as a controller. At each step t, the controller executes an action A_t, which samples a new architecture from the search space. It then observes the state S_t and receives a reward R_t from the environment. The environment involves training the sampled architecture using standard procedures and evaluating its performance, usually based on accuracy [26].

The controller's policy is updated based on the received rewards to improve the quality of the sampled architectures over time. Early work by Zoph et al. [46] used the policy gradient algorithm to train the controller, while later methods, such as those in [15], employed Proximal Policy Optimization (PPO) to refine the architecture search process.

3) Gradient Descent: Traditional neural architecture search strategies sample architectures from a discrete search space. However, DARTS (Differentiable Architecture Search) [47] introduced a pioneering approach by using Gradient Descent (GD) to explore a continuous and differentiable search space. This is achieved by relaxing the discrete space through a softmax function [47]:

$$o_{i,j}(x) = \sum_{k=1}^{K} \frac{\exp(\alpha_{i,j}^k)}{\sum_{k'=1}^{K} \exp(\alpha_{i,j}^{k'})} o^k(x), \tag{13}$$

where $o_{i,j}(x)$ represents the operation applied between nodes i and j, K is the number of possible operations, and $\alpha_{i,j}^k$ are the architecture parameters learned during the search. By optimizing these parameters with gradient descent, DARTS effectively searches for optimal architectures within a continuous space.

To manage the occurrence of specific operations like skip connections, P-DARTS introduces operation-level dropout. This technique regularizes the search space by controlling the frequency of skip connections during both training and evaluation, enhancing the robustness and efficiency of the search process [48].

4) Surrogate Model-based Optimization: Surrogate model-based optimization (SMBO) techniques have emerged as efficient methods for architecture optimization by leveraging past evaluation data to build predictive models of the objective function [4]. These techniques enhance search efficiency and reduce computational time. SMBO approaches typically use Bayesian optimization (BO) methods like Gaussian processes (GP), random forests (RF), and tree-structured Parzen estimators (TPE) [49]. However, GP-based methods face scalability issues with variable-length neural networks due to cubic time complexity in the number of observations. To address this, fixed-length encoding schemes using RF have been proposed.

VI. Model Evaluation

A. Low fidelity

In model evaluation, low fidelity methods are used to quickly estimate the performance of a model with reduced computational cost. These methods provide an approximate evaluation without requiring full training and evaluation cycles. One common low fidelity method is to use a smaller proxy dataset. For example, instead of training on the full ImageNet dataset, a smaller dataset like CIFAR-10 can be used to evaluate potential architectures. This significantly reduces the computational resources required:

$$\text{Resource Saving} = \frac{\text{Resources for Full Dataset}}{\text{Resources for Proxy Dataset}}. \tag{14}$$

It collectively help in evaluating models more efficiently, providing a good balance between accuracy and computational expense. In the context of learning transferable architectures, such as those discussed by Zoph et al. [27], these low fidelity evaluations allow for rapid iteration and refinement of neural network designs before committing to full-scale training and evaluation.

B. Weight sharing

Weight sharing is an efficient technique in model evaluation that significantly reduces the computational cost of training multiple neural network architectures. Instead of training each architecture from scratch, weight sharing allows multiple architectures to share a common set of weights for certain layers. This means that during the architecture search process, different network configurations reuse the same weights, leading to a substantial reduction in the number of parameters that need to be trained.

Mathematically, let $\mathcal{A}$ be the set of all possible architectures, and let $\mathcal{W}$ be the set of shared weights. For a given architecture $a \in \mathcal{A}$, its performance $P(a)$ is evaluated using the shared weights $\mathcal{W}$:

$$P(a) = f(a, \mathcal{W}), \tag{15}$$

where f represents the forward pass of the architecture a with the shared weights $\mathcal{W}$.

For example, in the NAS (Neural Architecture Search) framework, weight sharing is implemented by training a single

supernet that encompasses all candidate architectures [50]. Each candidate architecture samples a subset of the supernet's weights during its evaluation. This technique is particularly useful in large-scale architecture search problems where evaluating every possible architecture independently would be computationally prohibitive.

C. Surrogate

In hyperparameter optimization, evaluating new optimization techniques on real-world problems can be computationally expensive. To address this, surrogate models are employed. A surrogate model is a cheaper-to-evaluate approximation of the true performance function of a machine learning algorithm with respect to its hyperparameters [51].

The process involves training a regression model f to predict the performance y of a machine learning algorithm based on a set of hyperparameters $\mathbf{x}$:

$$y \approx f(\mathbf{x}), \tag{16}$$

where $\mathbf{x}$ represents the hyperparameter vector, and y is the observed performance metric (e.g., accuracy, error rate).

For instance, in the context of Bayesian optimization, the surrogate model can be used to guide the search for optimal hyperparameters by providing cheap and informative performance estimates [51]. This is especially useful in sequential model-based optimization, where the model is iteratively updated with new data points:

$$\mathbf{x}_{\text{new}} = \arg\max_{\mathbf{x}} \ \text{AcquisitionFunction}(\mathbf{x}; f), \tag{17}$$

where the acquisition function determines the next set of hyperparameters to evaluate based on the surrogate model f.

By leveraging surrogate models, we can achieve a balance between optimization efficiency and performance accuracy, making hyperparameter tuning more feasible and effective.

D. Early stopping

Early stopping is a regularization technique used to prevent overfitting in machine learning models, particularly in neural networks. The typical workflow for early stopping involves the following steps:

1) Split the dataset into training and validation sets.
2) Train the model on the training set.
3) Periodically evaluate the model on the validation set.
4) Track the validation performance, and stop training when the performance does not improve for a specified number of epochs.

The primary advantage of early stopping is that it reduces the risk of overfitting, allowing the model to generalize better to unseen data. It also saves computational resources by stopping the training process early when further training is unlikely to yield better results.

For example, consider the work by Klein et al. (2017) [52] which discusses using Bayesian neural networks for learning curve prediction to facilitate early stopping. They propose a model to predict the performance of neural networks over time and use these predictions to terminate poorly performing runs early, thereby speeding up the hyperparameter optimization process.

VII. NAS Discussion

In Table I, we compare various Neural Architecture Search (NAS) algorithms and their models, ranging from manually designed architectures to automated evolutionary search and reinforcement learning methods. Models like ResNet-110 [53] showcase the outstanding performance of manually designed architectures, despite limitations in parameters and computational resources. In contrast, models using evolutionary algorithms (EA), such as Large-scale ensemble and Hierarchical-EAS [28], [29], exhibit significant improvements in both parameter count and computational complexity while demonstrating higher accuracy.

Additionally, AmoebaNet-B and NASNet-A [27], [33], which utilize evolutionary search and reinforcement learning (RL), achieve new heights in performance and excel in parameter optimization and architecture generation. Particularly, NASNet-A stands out for its efficient optimization of GPU resources and training time. ENAS+macro [26] and DARTS [47] demonstrate the potential of optimization through reinforcement learning and gradient descent (GD), with DARTS notably reducing computational costs through differentiable architecture search.

Overall, as NAS methods continue to advance, these automated techniques exhibit significant advantages across different dimensions, providing crucial references for future deep learning model design and optimization.

VIII. Problems Existence and Future Directions

A. Interpretability

Differentiable Architecture Search (DARTS) has revolutionized the field of Neural Architecture Search (NAS) by significantly reducing the computational costs associated with designing neural networks [54]. However, despite its success, several challenges remain, particularly concerning the interpretability of the search process and the resulting architectures. Understanding why certain architectures perform better than others and how the search process can be improved remains a critical area of research.

B. Reproducibility

One of the significant challenges in machine learning (ML) is reproducibility, and this is particularly true for AutoML and Neural Architecture Search (NAS). The reproducibility issue arises because NAS algorithms typically require a substantial number of parameters to be manually set during implementation. Unfortunately, many original papers do not provide sufficient details on these parameter settings, including the random seed values used in experiments, which are crucial for replicating results [55].

TABLE I
COMPARISON OF NAS ALGORITHMS AND MODELS.

Reference	Published in	# Params (Millions)	Top-1/5 Acc (%)	GPU Days	# GPUs	AO
ResNet-110	ECCV16	1.7	93.57	-	-	Manually designed
PyramidNet	CVPR17	26.0	96.69	-	-	Manually designed
DenseNet	CVPR17	25.6	96.54	-	-	Manually designed
GeNet	ICCV17	-	92.9	17	-	Manually designed
Large-scale ensemble	ICLR17	40.4	95.6	-	-	EA
Hierarchical-EAS	ICLR18	15.7	96.25	2500	200	EA
AmoebaNet-B (N=6, F=128)	AAAI19	34.9	97.87	3150	450 K40	EA
NASNet-A ([6 @ 768]+c/o)	CVPR18	3.3	97.35	2000	500 P10	EA
ENAS+macro	ICML18	38.0	96.13	0.32	1	RL
DARTS	ICLR19	3.3	97.00	1.5	1	GD

C. Joint Hyperparameter and Architecture Optimization

Most Neural Architecture Search (NAS) studies have traditionally treated Hyperparameter Optimization (HPO) and Architecture Optimization (AO) as distinct processes. However, there is significant overlap between the methods used in both HPO and AO, such as Random Search (RS), Bayesian Optimization (BO), and Gradient-based Optimization (GO). This overlap suggests that it is feasible to jointly optimize hyperparameters and architectures, an approach supported by several recent studies.

One notable example of successful joint Hyperparameter and Architecture Optimization (HAO) is demonstrated in the work on FBNetV3: Joint Architecture-Recipe Search using Predictor Pretraining by Dai et al. (2020) [56]. In this study, the authors propose Neural Architecture-Recipe Search (NARS), which simultaneously optimizes both the network architecture and its training hyperparameters. They utilize an accuracy predictor that scores both architecture and training recipes together, guiding the search process to find the best combinations. This method significantly improves the efficiency of the search and results in state-of-the-art performance for various tasks.

IX. CONCLUSION

In this paper, we provide a thorough review of AutoML studies from data preparation to model evaluation. We compare current NAS algorithms, offering a clear understanding of AutoML, supporting future research direction in this field.

REFERENCES

[1] M.-A. Zöller and M. F. Huber, "Benchmark and survey of automated machine learning frameworks," *Journal of artificial intelligence research*, 2021.

[2] X. He, K. Zhao, and X. Chu, "Automl: A survey of the state-of-the-art," *Knowledge-based systems*, 2021.

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

[4] M. Feurer, A. Klein, K. Eggensperger, J. Springenberg, M. Blum, and F. Hutter, "Efficient and robust automated machine learning," *Advances in neural information processing systems*, vol. 28, 2015.

[5] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial networks," *Communications of the ACM*, 2020.

[6] A. Mikołajczyk and M. Grochowski, "Data augmentation for improving deep learning in image classification problem," in *2018 international interdisciplinary PhD workshop (IIPhDW)*. IEEE, 2018.

[7] ——, "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.

[8] A. Antoniou, A. Storkey, and H. Edwards, "Data augmentation generative adversarial networks," *arXiv preprint arXiv:1711.04340*, 2017.

[9] S. C. Wong, A. Gatt, V. Stamatescu, and M. D. McDonnell, "Understanding data augmentation for classification: when to warp?" in *2016 international conference on digital image computing: techniques and applications (DICTA)*. IEEE, 2016.

[10] R. Hataya, J. Zdenek, K. Yoshizoe, and H. Nakayama, "Faster autoaugment: Learning augmentation strategies using backpropagation," in *Computer Vision–ECCV 2020: 16th European Conference, Glasgow, UK, August 23–28, 2020, Proceedings, Part XXV 16*. Springer, 2020.

[11] S. Lim, I. Kim, T. Kim, C. Kim, and S. Kim, "Fast autoaugment," *Advances in Neural Information Processing Systems*, vol. 32, 2019.

[12] C. Lin, M. Guo, C. Li, X. Yuan, W. Wu, J. Yan, D. Lin, and W. Ouyang, "Online hyper-parameter learning for auto-augmentation strategy," in *Proceedings of the IEEE/CVF international conference on computer vision*, 2019.

[13] M. Geng, K. Xu, B. Ding, H. Wang, and L. Zhang, "Learning data augmentation policies using augmented random search," *arXiv preprint arXiv:1811.04768*, 2018.

[14] S. Krishnan, J. Wang, M. J. Franklin, K. Goldberg, T. Kraska, T. Milo, and E. Wu, "Sampleclean: Fast and reliable analytics on dirty data." *IEEE Data Eng. Bull.*, 2015.

[15] S. Krishnan, M. J. Franklin, K. Goldberg, J. Wang, and E. Wu, "Activeclean: An interactive data cleaning framework for modern machine learning," in *Proceedings of the 2016 International Conference on Management of Data*, 2016.

[16] S. Krishnan, M. J. Franklin, K. Goldberg, and E. Wu, "Boostclean: Automated error detection and repair for machine learning," *arXiv preprint arXiv:1711.01299*, 2017.

[17] S. Krishnan and E. Wu, "Alphaclean: Automatic generation of data cleaning pipelines," *arXiv preprint arXiv:1904.11827*, 2019.

[18] I. Gemp, G. Theocharous, and M. Ghavamzadeh, "Automated data cleansing through meta-learning," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 31, no. 2, 2017.

[19] H. Vafaie and K. De Jong, "Evolutionary feature space transformation," in *Feature Extraction, Construction and Selection: a data mining perspective*. Springer, 1998.

[20] Q. Meng, D. Catchpoole, D. Skillicom, and P. J. Kennedy, "Relational autoencoder for feature extraction," in *2017 International joint conference on neural networks (IJCNN)*. IEEE, 2017.

[21] O. Irsoy and E. Alpaydın, "Unsupervised feature extraction with autoencoder trees," *Neurocomputing*, 2017.

[22] C. Thornton, F. Hutter, H. H. Hoos, and K. Leyton-Brown, "Auto-weka: Combined selection and hyperparameter optimization of classification algorithms," in *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2013.

[23] C. Cortes and V. Vapnik, "Support-vector networks," *Machine learning*, 1995.

[24] N. S. Altman, "An introduction to kernel and nearest-neighbor nonparametric regression," *The American Statistician*, 1992.

[25] S. I. Gallant *et al.*, "Perceptron-based learning algorithms," *IEEE Transactions on neural networks*, 1990.

[26] H. Pham, M. Y. Guan, B. Zoph, Q. V. Le, and J. Dean, "Efficient neural architecture search via parameter sharing," *Proceedings of the 35th International Conference on Machine Learning (ICML)*, 2018.

[27] B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le, "Learning transferable architectures for scalable image recognition," *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018.

[28] H. Liu, K. Simonyan, O. Vinyals, C. Fernando, and K. Kavukcuoglu, "Hierarchical representations for efficient architecture search," in *Proceedings of the 6th International Conference on Learning Representations (ICLR)*. OpenReview.net, 2018.

[29] C. Liu, L.-C. Chen, F. Schroff, H. Adam, W. Hua, A. Yuille, and L. Fei-Fei, "Auto-deeplab: Hierarchical neural architecture search for semantic image segmentation," *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019.

[30] J. Guo, K. Han, Y. Wang, C. Zhang, Z. Yang, H. Wu, X. Chen, and C. Xu, "Hit-detector: Hierarchical trinity architecture search for object detection." IEEE, 2020.

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

[32] T. Wei, C. Wang, Y. Rui, and C. W. Chen, "Network morphism," in *Proceedings of the 33rd International Conference on Machine Learning (ICML)*. ACM, 2016.

[33] E. Real, A. Aggarwal, Y. Huang, and Q. V. Le, "Regularized evolution for image classifier architecture search," in *Proceedings of the Thirty-Third AAAI Conference on Artificial Intelligence (AAAI)*. AAAI Press, 2019, pp. 4780–4789.

[34] A. Kwasigroch, M. Grochowski, and M. Mikolajczyk, "Deep neural network architecture search using network morphism." IEEE, 2019.

[35] C.-W. Hsu, C.-C. Chang, C.-J. Lin *et al.*, "A practical guide to support vector classification," 2003.

[36] J. Y. Hesterman, L. Caucci, M. A. Kupinski, H. H. Barrett, and L. R. Furenlid, "Maximum-likelihood estimation with a contracting-grid search algorithm," *IEEE transactions on nuclear science*, 2010.

[37] J. Bergstra and Y. Bengio, "Random search for hyper-parameter optimization." *Journal of machine learning research*, vol. 13, no. 2, 2012.

[38] S. Falkner, A. Klein, and F. Hutter, "Bohb: Robust and efficient hyperparameter optimization at scale," in *International conference on machine learning*. PMLR, 2018.

[39] A. Klein, S. Falkner, S. Bartels, P. Hennig, and F. Hutter, "Fast bayesian optimization of machine learning hyperparameters on large datasets," in *Artificial intelligence and statistics*. PMLR, 2017.

[40] D. Maclaurin, D. Duvenaud, and R. Adams, "Gradient-based hyperparameter optimization through reversible learning," in *International conference on machine learning*. PMLR, 2015.

[41] L. Franceschi, M. Donini, P. Frasconi, and M. Pontil, "Forward and reverse gradient-based hyperparameter optimization," in *International Conference on Machine Learning*. PMLR, 2017.

[42] K. Chandra, A. Xie, J. Ragan-Kelley, and E. Meijer, "Gradient descent: The ultimate optimizer," *Advances in Neural Information Processing Systems*, 2022.

[43] L. Xie and A. L. Yuille, "Genetic cnn," in *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*. IEEE Computer Society, 2017.

[44] C. Fernando, D. Banarse, M. Reynolds, F. Besse, D. Pfau, M. Jaderberg, M. Lanctot, and D. Wierstra, "Convolution by evolution: Differentiable pattern producing networks." ACM.

[45] C. e. a. a. g. g. F. Gruau, "Convolution by evolution: Differentiable pattern producing networks," in *Colloquium on Grammatical Inference*. IEEE.

[46] B. Zoph and Q. V. Le, "Neural architecture search with reinforcement learning," in *Proceedings of the 5th International Conference on Learning Representations (ICLR)*. OpenReview.net, 2017.

[47] H. Liu, K. Simonyan, and Y. Yang, "Darts: Differentiable architecture search," in *Proceedings of the 7th International Conference on Learning Representations (ICLR)*. OpenReview.net, 2019.

[48] X. Chen, L. Xie, J. Wu, and Q. Tian, "Progressive differentiable architecture search: Bridging the depth gap between search and evaluation," in *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*. IEEE, 2019.

[49] R. Negrinho, M. R. Gormley, G. J. Gordon, D. Patil, N. Le, and D. Ferreira, "Towards modular and programmable architecture search," in *Advances in Neural Information Processing Systems 32 (NeurIPS)*, H. M. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché Buc, E. B. Fox, and R. Garnett, Eds., 2019, pp. 13 715–13 725.

[50] C. Wong, N. Houlsby, Y. Lu, and A. Gesmundo, "Neural architecture search with reinforcement learning," *Advances in neural information processing systems*, 2018.

[51] K. Eggensperger, F. Hutter, H. H. Hoos, and K. Leyton-Brown, "Efficient benchmarking of hyperparameter optimizers via surrogates," *Journal of Artificial Intelligence Research*, 2014.

[52] A. Klein, S. Falkner, J. T. Springenberg, and F. Hutter, "Learning curve prediction with bayesian neural networks," *Proceedings of the 5th International Conference on Learning Representations (ICLR)*, 2017.

[53] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE Computer Society, 2016, pp. 770–778.

[54] A. Zela, T. Elsken, T. Saikia, Y. Marrakchi, T. Brox, and F. Hutter, "Understanding and robustifying differentiable architecture search," *Proceedings of the 8th International Conference on Learning Representations (ICLR)*, 2020.

[55] L. Li and A. Talwalkar, "Random search and reproducibility for neural architecture search," in *Proceedings of the Thirty-Fifth Conference on Uncertainty in Artificial Intelligence (UAI)*, A. Globerson and R. Silva, Eds., vol. 115. AUAI Press, 2019, pp. 367–377.

[56] X. Dai, A. Wan, P. Zhang, B. Wu, Z. He, Z. Wei, K. Chen, Y. Tian, M. Yu, P. Vajda, and J. E. Gonzalez, "Bnetv3: Joint architecture-recipe search using predictor pretraining," *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2021.