Techniques for Automated Machine Learning

Yi-Wei Chen, Qingquan Song, Xia Hu
Department of Computer Science and Engineering
Texas A&M University
{yiwei_chen, song_3134, xiahu}@tamu.edu

ABSTRACT

Automated machine learning (AutoML) aims to find optimal machine learning solutions automatically given a machine learning problem. It could release the burden of data scientists from the multifarious manual tuning process and enable the access of domain experts to the off-the-shelf machine learning solutions without extensive experience. In this paper, we review the current developments of AutoML in terms of three categories, automated feature engineering (AutoFE), automated model and hyperparameter learning (AutoMHL), and automated deep learning (AutoDL). State-of-the-art techniques adopted in the three categories are presented, including Bayesian optimization, reinforcement learning, evolutionary algorithm, and gradient-based approaches. We summarize popular AutoML frameworks and conclude with current open challenges of AutoML.

1. INTRODUCTION

Automated machine learning (AutoML) has emerged as a prevailing research field upon the ubiquitous adoption of machine learning techniques. It aims at automatically determining high-performance machine learning solutions with a little workforce in reasonable time budget. For example, Google HyperTune, Amazon Model Tuning, and Microsoft Azure AutoML all provide cloud services with AutoML tools which cultivate off-the-shelf machine learning solutions for both researchers and practitioners. Therefore, AutoML not only liberates them from the time-consuming tuning process and tedious trial-and-error iterations but also facilitates the development of solving machine learning problems.

A traditional machine learning pipeline is an iterative procedure composed of feature engineering, model and hyperparameter selection, and performance evaluation, as shown in Figure 1. Data scientists manually manipulate numerous features, design models, and tune hyperparameters in order to get the desired predictive performance. The procedure will not be terminated until a satisfactory performance is achieved. Echoing with the traditional pipeline, we split AutoML into three categories, (1) AutoFE: automated feature engineering, (2) AutoMHL: automated model and hyperparameter learning, and (3) AutoDL: automated deep learning. AutoFE intends to discover informative and discriminative features for the learning model. AutoMHL aims at tuning hyperparameter of a specific learning model (HPO) or building an entire machine learning pipeline automatically (Auto-Pipeline). AutoDL is a subfield of AutoMHL. Since deep learning has succeeded significantly without intensive feature engineering, we explicitly separate AutoDL from AutoMHL in order to focus on the automatic design of deep neural network architectures. Upon the categorization of AutoML, we will review AutoML from two dimensions (Figure 2). The technique dimension spans the mainstream AutoML techniques, including Bayesian optimization (BO), reinforcement learning (RL), evolutionary algorithm (EA), and gradient-based approaches (Gradient), while the framework dimension covers representative AutoML frameworks.

The remainder of the paper is organized as follows: Section 2 discusses AutoFE with RL, EA, and other techniques. Section 3 explains AutoMHL with BO, RL, and EA. Section 4 illustrates AutoDL in terms of BO, RL, and Gradient. Section 5 explores the emblematic open-source and enterprise AutoML frameworks. The last two sections specify current research challenges and reemphasize the target of this work. Table 2.3 summarizes the AutoML techniques mentioned in the work.

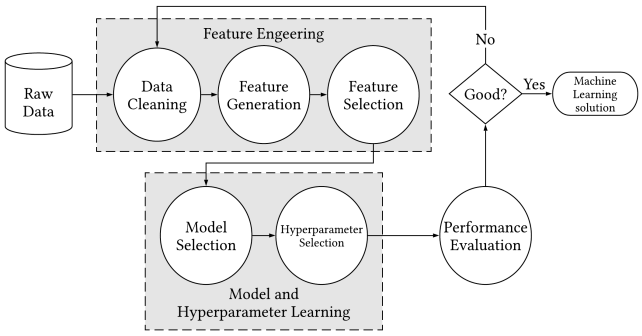

Figure 1: A classical machine learning pipeline. When data scientists intend to find their machine learning solutions, they manually try the combination of features, models, and hyperparameters in a pipeline and repeatedly tune its performance until the final solution achieves good general prediction. AutoML is proposed to facilitate the labor-intensive tuning process.

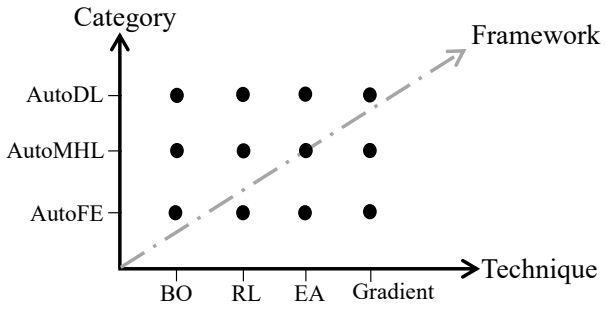

Figure 2: A taxonomy of AutoML from techniques and frameworks. We split AutoML into automated feature engineering (AutoFE), automated model and hyperparameter learning (AutoMHL), and automated deep learning (AutoDL). The three categories are illustrated from the techniques dimension, including Bayesian optimization (BO), reinforcement learning (RL), evolutionary algorithm (EA), and gradient-based approaches (Gradient). Selected AutoML frameworks are discussed beyond.

2. AUTOMATED FEATURE ENGINEERING

Feature engineering is a process to manipulate features via operations such as data imputation, feature transformation, and feature selection. It is a critical step in machine learning algorithms since suitable features directly influence their prediction performance [5]. Considering a dataset $\mathcal{D}^F$ represented in terms of its feature set F, the derived features $\hat{F}$ in $\mathcal{D}^{\hat{F}}$ are generated by applying feature pre-processing operations $\mathcal{T}$ to $\mathcal{D}^F$. The automated feature engineering could be formulated in the following optimization problem,

$$\begin{aligned} F^* &= \arg\min_{F \in \mathcal{F} \cup \hat{\mathcal{F}}} \mathcal{L}_{val}(A_{w^*}, D^F_{val}), \\ \text{s.t.} \quad w^* &= \arg\min_{w} \mathcal{L}_{train}(A_w, D^F_{train}), \end{aligned} \tag{1}$$

where $\mathcal{L}_{train}$ and $\mathcal{L}_{val}$ are the loss function for training and validation, D^F_{train} and D^F_{val} denote training and validation set from D^F, and w is the parameters (weights) of a machine learning algorithm A. We intend to pick up vigorous features that can boost the performance of A through AutoFE. Generally, AutoFE follows the iterative procedure to find the optimal features, including two necessary phases, (1) feature generation and (2) feature selection. The original features are extended by feature generation and then feature selection will filter irrelevant and overlapping features. In this section, we briefly introduce two popular techniques in AutoML, i.e., reinforcement learning (RL) and evolutionary algorithm (EA), under the context of AutoFE.

2.1 Reinforcement Learning

Reinforcement learning (RL) is one of the most popular techniques used in AutoML. The goal of RL is to train an agent that could learn how to take a sequence of proper actions to maximize the long-term reward in the specific environment. Algorithm 1 illustrates the iterative procedure of RL. The agent takes actions upon a particular strategy (a.k.a. policy) to learn experience for correct decisions. The environment produces the reward based on the action and the current state of the agent. The agent, the environment, actions, and the reward are essential elements of RL. In the context of AutoML, the environment is the generated machine learning solution, the reward corresponds to its performance on the holdout dataset, and the actions are operations to modify a learning solution.

Algorithm 1: Reinforcement Learning

Input : The policy π; environment *env*, including the reward function;
Output: The well-trained policy π

```
Initialize state s_i and policy π
while not meet stopping criteria do
    action a* = arg max_a π(s_i, a)
    s_{i+1}, r = env(s_i, a*)
    update π with s_{i+1}, r
    i = i + 1
end
return π
```

Within the context of AutoFE, the environment corresponds to all the possible combinations of features and transformations, which could be represented in a transformation graph [18] (Figure 3). The initial node of the graph is the original feature set. Edges are feature transformations (e.g., square, sum, and log), and feature selection. Other nodes are the derived features. Note that the transformation is applied to all features regardless of feature types. In the transformation graph, the traversal from the initial node to an end node is a path of feature transformation that applies all feature operators to the original dataset. The goal is to learn the traversal within the limited time budge to reach the optimal node brings about the highest performance.

The agent can learn the graph exploration with the help of Q-Learning, which uses Q function, as shown in Equation (2) to remember previous learning experiences.

$$Q(s^{(t)}, a^{(t)}) \leftarrow (1-\eta)Q(s^{(t)}, a^{(t)}) + \eta[r^{(t)} + \gamma \max_{a'} Q(s^{(t+1)}, a')], \tag{2}$$

where s, a, η, and γ correspond to state, action, learning rate, and discount factor respectively and $r^{(t)}$ denotes the reward of step t. An action of the agent is a pair of an existing node n in the current graph and a feature operation t to the node. The ϵ-greedy algorithm determines the next action, i.e., (n, t) is chosen randomly with probability ϵ or from the maximum of Q function with probability $1 - \epsilon$. In each step, an action produces new features with which the learning algorithm A is trained to get the validation performance. The performance serves as the reward r for the Q-function. Also, the current graph is augmented with (n, t) and r.

Since the combination of states and feature operations grows exponentially in the graph, Q-Learning with linear approximation [18] is proposed. That is, Q function approximates in $Q(s, a) = w^a f(s)$, where w^a is a weight vector for an action a, and $f(s)$ is a state vector characterizing the state s. w^a is updated by the following equation,

$$w^{a_t} \leftarrow w^{a_t} + \alpha(r^{(t)} + \gamma \max_{a'}[Q(s^{(t+1)}, a') - Q(s^{(t)}, a)])f^{(t)}(s) \tag{3}$$

where α is the learning rate. As long as the Q function is learned, the agent can decide the exploration direction.

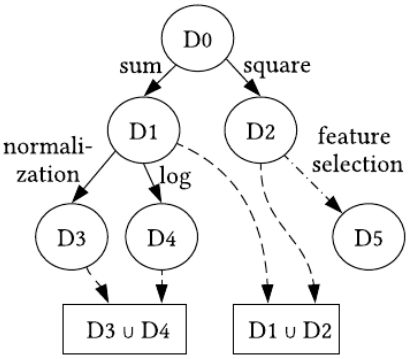

Figure 3: The transformation graph for AutoFE [18]. D_0 is the initial dataset. The feature engineering operations such as sum, square, log, normalization, and feature selection, are used in the graph.

Algorithm 2: Evolutionary Algorithm

Input : Individual representation; fitness measurement $\mathcal{M}$; Selection; Partition; Crossover; Mutation
Output: The optimal individual

```
Initialize a population of random individuals, P
while not meet stopping criteria do
    for individual i in P do
        M(i) ; // evaluate fitness of individual
    end
    Generation G = Selection(P)
    G_1, G_2 = Partition(G) ; // split G to two sub-groups
    P = Crossover(G_1) ∪ Mutation(G_2)
end
return The individual with the highest fitness
```

2.2 Evolutionary Algorithm

Evolutionary algorithm imitates the natural evolution to solve optimization problems, including genetic algorithms (GAs) and genetic programming (GP). We will take AutoML using GP as an example in this survey, while other similar evolutionary algorithms such as particle swarm optimization (PSO) and ant colony optimization (ACO) will not be covered due to the limited space.

Typically, EA follows Algorithm 2, including population, selection, crossover, and mutation. In the beginning, the evolution algorithms randomly generate individuals as the initial population. After evaluating the population, selection methods (e.g., tournament and elitism selection) will pick up competent individuals as parents to form the next population. In the mutation (crossover) step, mutation operations change parents' structure to create various individuals in the new population. The selection-mutation-crossover will be repeated until a particular condition is satisfied (e.g., a fixed number of generations).

Equipped AutoFE with EA, constructed features could be represented in the tree structure [30] where the root is a derived feature, the intermediate nodes are arithmetic operators, and leaves are either original features or random values. It is worth pointing out that the root of the tree is different from the transformation graph [18], where the starting node is the original feature set. Genetic programming modifies the intermediate nodes and leaves of a tree-based individual to generate new features in the mutation and crossover phase. The selection phase of GP implicitly picks up candidate features. Therefore, the feature generation and selection are executed together within GP [30].

2.3 Other Methods

Besides RL and EA, there are also other approaches [14, 15, 19, 16] for AutoFE. The structure of datasets leads to the various techniques of feature generation. Meanwhile, the performance evaluation of selected features brings about varying techniques of feature selection. We review them in terms of feature generation and feature selection.

For feature generation, we can categorize datasets to relational and non-relational. Relational datasets provide relationships between entities (tables) by following which we generate additional features [14]. We apply arithmetic operators to columns of one entity that do or do not depend on columns of another entity. For instance, we can obtain a new feature of average customer expense by averaging the "Order" entity based on the IDs of the"Customer" entity. When it comes to non-relational datasets, we directly utilize the features in the dataset. We can construct ridge and kernel ridge regression models to measure the feature correlations that serve as additional new features [16]. If datasets are expressed in the transformation graph (Figure 3), depth-first and breath-first traversals [19] without training an agent offer a simple way to decide the next node and feature transformation.

For feature selection, selected features could be evaluated by a tree model or a machine learning pipeline. We can train a decision tree to get the information gain of each feature and use the stability-based selection that aggregates the results of many selection algorithms on different subsets of features. So mixed results are used to select final features [16]. Another way is constructing a random forest (RF) model over meta-features between datasets and new generated features [15]. The rank of generated features from the output of the RF model is used to select features. Apart from tree-based selections, a fixed-structure machine learning pipeline could be constructed for evaluation, including Truncated SVD for feature selection [14]. We select n_c components from the results of SVD transformation, rank the components by calculating their f-value w.r.t. the targets, and keep top $\gamma\%$ of them. The hyperparameters n_c and γ are tuned by Bayesian optimization to select the number of features for the pipeline.

3. AUTOMATED MODEL AND HYPERPARAMETER LEARNING

Model and hyperparameter learning consists of model selection and hyperparameter tuning, which optimizes the predictive performance by repeatedly changing machine learning models and tuning associate hyperparameter values. Given a dataset $\mathcal{D}$ divided into $\mathcal{D}_{train}$ and $\mathcal{D}_{val}$ for training and validation respectively, we should choose a loss function $\mathcal{L}_{train}$ for training (e.g., square mean error or cross entropy) and a measurement $\mathcal{L}_{val}$ for validation (e.g., accuracy or F1-score). Considering a set of learning models $\mathcal{A} = \{A^{(1)}, A^{(2)}, ...\}$,

Category \ Technique	Bayesian Optimization	Reinforcement Learning	Evolutionary Algorithm	Gradient-based Approaches	Other techniques
Automated Feature Engineering (AutoFE)	-	FeatureRL [18]	GP for Feature Engineering [30]	-	Data Science Machine [14] ExploreKit [15] Cognito [19] AutoLearn [16]
Automated Model and Hyperparameter Learning (AutoMHL)	TPE [1] SMAC [10] Auto-Sklearn [9] FABOLAS [20] BOHB [8]	APRL [17] Hyperband [21]	TPOT [25] Autostacker [3] DarwinML [27]	-	-
Automated Deep Learning (AutoDL)	AutoKeras [12] NASBot [13]	NAS [34] NASNet [35] ENAS [26]	-	DARTS [22] Proxyless [2] NAONet [23]	-

Table 1: Overview of different techniques for Automated Machine Learning.

automated model and hyperparameter learning (AutoMHL) could be formulated as the following optimization problem,

$$\begin{aligned} A^*_{\lambda^*} = & \underset{A^{(i)}\in\mathcal{A},\lambda\in\Lambda^{(i)}}{\arg\min} \mathcal{L}_{val}(A^{(i)}_{\lambda,w^*}, D_{val}), \\ \text{s.t.} \quad w^* = & \underset{w}{\arg\min}\, \mathcal{L}_{train}(A^{(i)}_{\lambda,w}, D_{train}), \end{aligned} \tag{4}$$

where $\Lambda^{(i)}$ denotes the hyperparameter space of $A^{(i)}$, w is the parameters of a model A. The goal is to obtain the optimal learning model A^* and its hyperparameters λ^*. If we reduce $\mathcal{A} = \{A^{(1)}\}$, AutoMHL becomes hyperparameter optimization (HPO). If we expand $\mathcal{A}$ to include features F_i, i.e., $\mathcal{A} = \{A^{(1)}, A^{(2)}, ...F_1, F_2, ...\}$, AutoMHL becomes the automated pipeline learning (Auto-Pipeline). Therefore, AutoMHL is the general form of the above two AutoML problems. In this section, we briefly introduce BO and review AutoMHL in terms of BO, RL, and EA.

3.1 Bayesian Optimization

Bayesian optimization (BO) is a well-known method to find the optimal value of a black-box and non-convex objective function. It contains the two indispensable components, a probabilistic model and an acquisition function. The probabilistic model is a surrogate model emulating the objective function f, which models the functional relationship between the search space and the objective function values. The acquisition function is constructed based on the surrogate to measure potential points in the search space by considering the trade-off between exploration (trying uncertain points) and exploitation (improving current good points). BO goes through an iterative procedure of three steps, as shown in Algorithm 3. (1) Line 4: obtain the next point to evaluate by optimizing the acquisition function. (2) Line 5: evaluate the point to get its function value. (3) Line 7: update the surrogate model with the new point and the value.

There are two challenges to be considered when applying BO for AutoMHL, including heterogeneous search space and efficiency. The first challenge results from various hyperparameters of a learning model, including discrete, categorical, continuous hyperparameters, and hierarchical dependency (conditional) among them. Take support vector machines (SVM) as an example. Its kernel, degree, and gamma is a categorical, discrete, and continuous hyperparameter respectively, while the degree also depends on polynomial kernel. The efficiency bottleneck usually arises from the performance evaluation step. In every iteration, we need to obtain the real value of the objective function. The situation becomes worse when it is costly to train a learning model (e.g., neural network) on a large-scale dataset. Therefore, various techniques have been proposed to address these challenges.

First, tree-based BO [10, 1] have emerged as alternates to handle the heterogeneous search space. Gaussian process in standard BO cannot consider the hierarchical dependency and model the probabilities of various types of hyperparameters. Random forest (RF) regression could replace it as the surrogate model [10]. Random subsets of historical observations build individual decision trees. One tree node decides the value to split a discrete/continuous hyperparameter or whether a hyperparameter is active to handle conditional variables. To build an acquisition function (e.g., Expected improvement) upon the RF regression, we have to measure the probability $p(c|\lambda)$ of real performance c given hyperparameters λ. The RF regression does not produce $p(c|\lambda)$, but the frequency estimates of μ_λ and σ^2_λ from individual trees could be used to approximate $p(c|\lambda) = \mathcal{N}(\mu_\lambda, \sigma^2_\lambda)$.

Another way to replace Gaussian process is by building a probability density tree [1], in which a tree node stands for the probability density of a specific hyperparameter rather than a decision. A hyperparameter serves as a random variable. There are two types of probability density in a node,

$$p(\lambda|c) = \begin{cases} l(\lambda) = p(\lambda|c < c^*) \\ g(\lambda) = p(\lambda|c >= c^*). \end{cases} \tag{5}$$

The probability density of hyperparameters λ conditional on whether the performance c is less or greater than a given performance threshold c^*. For the dependent hyperparameters, only values of active ones update the probability of the corresponding nodes. Since the density tree uses $p(\lambda|c)$ instead of $p(c|\lambda)$, Bergstra et al. [1] show that the optimization of Expected improvement could be simplified by $\lambda^* = \arg\min_\lambda \frac{g(\lambda)}{l(\lambda)}$.

In the context of Auto-Pipeline, we can expand the search space [9] by adding pipeline operations, such as data imputation, feature scaling, transformation, and selection. The tree-based BO [10, 1] could find the optimal pipeline solution in the new search space. However, the new space is much larger than that of HPO. Considering past performance on similar datasets [9] enables BO to start in the small but reasonable regions, which reduces the number of intermediate results. For those sub-optimal pipelines during a search, we can also build a weighted ensemble of them [9] to get robust performance.

Algorithm 3: Bayesian optimization

Input : Objective function f; surrogate model $\mathcal{M}$; acquisition function α; current observations $\mathcal{D}_n = \{(\mathbf{x}_i, y_i)\}_{i=1}^n$

; **Output:** The optimal value of f

Initialize $\mathcal{M}$ and α

while *not meet criteria* **do**

  $\mathbf{x}_{n+1} = \arg\max_{\mathbf{x}} \alpha(\mathbf{x}; \mathcal{D}_n)$

  $y_{n+1} = f(\mathbf{x}_{n+1})$

  $\mathcal{D}_n = \mathcal{D}_n \cup \{(\mathbf{x}_{n+1}, y_{n+1})\}$

  update $\mathcal{M}$ with $\mathcal{D}_n$

end

return The optimal value of f according to $\mathcal{D}_n$

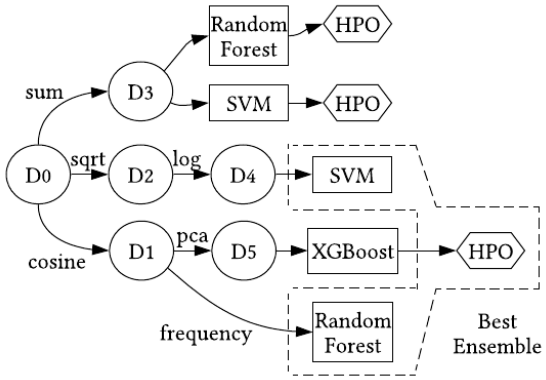

Figure 4: The exploration tree for AutoMHL with RL [17]. The root D_0 is the initial dataset, and circular nodes are derived datasets by applying a feature transformation. Rectangular nodes stand for machine learning algorithms. HPO indicates whether we tune hyperparameters of the learning model. The goal is to find the ensemble of learning models for a machine learning pipeline.

Second, approximate performance evaluation (multi-fidelity optimization) is proposed to decrease the evaluation cost. Specifically, we could train the model learning in an economical setting. For example, we can use a subset of training data [24] or fewer epochs of stochastic gradient descent [29] when training neural networks. Performance extrapolation is an alternative acceleration method by extrapolating the performance of the full dataset based on the historical records of the partial datasets. For instance, we can prepare two objective functions, the loss function for performance evaluation and the training time function, both of which consider the size of the training set as an additional input [20]. In individual iterations, Bayesian optimizer determines the proper size of training data that could take a few training time and ideally represent the performance of the full training set as well. Thus, the approach could allow training the learning model with a small dataset in early stages and extrapolating the performance of the full set.

Dynamic resources assignment is an approach assigning more training budgets for promising configurations and fewer budgets for discouraging configurations. Hyperband [21] is a representative example illustrated in Section 3.2. An incremental work BOHB [8] using Hyperband to speed up a tree-based BO approach [1]. Different from Hyperband [21] sampling configurations by random search, BOHB relies on a model-based approach (TPE) [1] to select encouraging configurations, making it easier to converge the performance of resulting configurations.

3.2 Reinforcement Learning

Speaking of AutoMHL with RL, the agent is required to determine the hyperparameters of a learning algorithm (HPO) [21] or design a machine learning pipeline (Auto-Pipeline) [17]. We review the RL techniques from the aspect of HPO and Auto-Pipeline.

For HPO, we could frame it in the multi-armed bandit, a particular case of RL. The environment consists of multiple arms, and the agent pulls a sequence of the arms to obtain the maximal reward. A set of hyperparameters stands for an arm whose reward is the performance of a learning model with the hyperparameters [21]. The goal is to find the optimal arm by pulling a sequence of arms one-by-one. Given n arms (n sets of hyperparameters), we can quickly determine the optimal one as long as getting their performance. To efficiently obtain the performance under a fixed budgets B, successive halving [11] assigns each configuration $\frac{B}{n}$ budgets and then select the best half with double budgets in the next iteration until one configuration remains. Furthermore, we could dynamically adjust budgets (e.g., the number of iterations) according to n [21]; large n configurations occupy few budgets while few n configurations consume large budgets. The two-fold dynamic resource allocation significantly reduces the evaluation time [21]. Thus, when n arms are sampled by random search [21] or tree-based BO [8], the aforementioned multi-armed techniques could efficiently determine the best one.

For Auto-Pipeline, we could picture its search space to the exploration tree [17] (Figure 4), where the root is the original dataset with raw features, intermediate nodes are derived features after feature transformation, and the leaves are HPO or learning models. HPO indicates whether to tune hyperparameters of the parent node. The agent selects an existing node and applies a transformation, a learning model, or HPO to the node. The goal of RL is to learn an exploration policy in the tree so that selected learning models could form an ensemble providing the best predictive performance. The agent takes advantage of Q-Learning to remember historical rewards and adopts ϵ-greedy algorithm to decide the next node and the operation. To avoid the exponential combinations of nodes and operations, Q-function is approximated by $Q(s, c) = w * f(s)$ [17], where s, c, and w denote a state, an action, and a weight vector respectively. $f(s)$ is a state vector characterizing the state s. Following the Equation (3), we can learn the Q-function to decide actions in the tree.

3.3 Evolutionary Algorithm

Extant AutoMHL work with EA [25, 3, 27] focus on automated pipeline learning (Auto-Pipeline). In particular, they utilize genetic programming (GP) to search machine learning pipelines. GP encodes pipelines in flexible graph structures and evolves them into the optimal one that provides the highest predictive performance. There are three types of flexible pipeline structures, tree-based [25], layer-based [3], and graph-based pipeline [27].

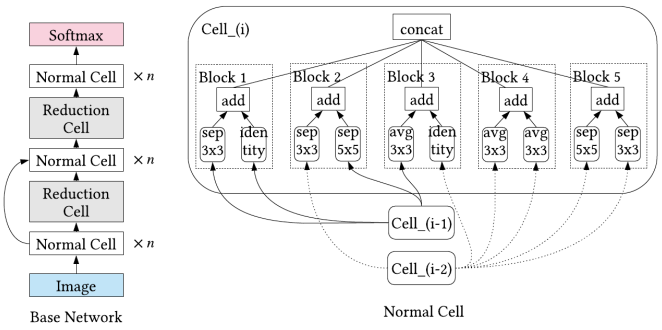

Figure 5: An example of a cell-based search space [35]. The left figure is a base network that stacks normal cells and reduction cells with skip connections. The right one is a normal cell, a convolutional cell that returns a feature map of the same dimension. A reduction cell is also a convolutional cell with a large stride size, which might have the same architecture as the normal cell. Both types of cell are composed of multiple blocks where each block has two operations and two inputs, and these operation results are added. The sep, avg, and identity denote depthwise-separable convolution, average pooling, and 1×1 convolution, respectively. In the search space, we are supposed to determine the two cell architectures and construct the base network to evaluate the performance of resulting cells.

First, in a tree-based structure [25], leaves are the copies of input data, intermediate nodes are pipeline operators, such as feature pre-processing methods, feature selection algorithms, and machine learning models and the root node must be a learning model. A tree pipeline has one or more pre-processing nodes followed by a selection node and a model node. The mutation and crossover of GP might change the types of pipeline operators as well as hyperparameters of each pipeline operators. The fitness is the predictive performance of a model node after intermediate nodes process input data.

Second, a layer-based pipeline [3] is similar to multiple layer perceptron, but each node is a machine learning model. The first layer is the input data, and the outputs of the following layer are new synthetic features added to the raw features that serve as the input of the next layer. The search space contains the number of layers, the number of nodes per layer, types of learning models, and associate hyperparameter of each learning model. GP tunes the above space to find the optimal pipeline.

Finally, the most general pipeline structure is a directed acyclic graph (DAG) [27], where edges and vertices indicate data flow and learning model respectively. A pipeline is able to combine many various types of machine learning models, such as the mix of classifiers, regressors, and unsupervised learning models. The adjacent matrix controls the edge connections of a DAG, which is evolved by mutation operators. After GP designs the pipeline structure and learning models, associate hyperparameters could be tuned by Bayesian optimization. The last two pipeline structures do not explicitly operate feature engineering, but they leverage outputs of learning models as new features.

4. AUTOMATED DEEP LEARNING

Automated deep learning (AutoDL) is purposed to facilitate the design of neural architectures and the selection of their hyperparameters. It can be regarded as a particular topic of AutoMHL. Following Equation (4), we can describe AutoDL in the following optimization problem,

$$\begin{aligned} A^* &= \arg\min_{A \in \mathcal{A}} \mathcal{L}_{val}(A(w^*), D_{val}), \\ \text{s.t.} \quad w^* &= \arg\min_{w} \mathcal{L}_{train}(A(w), D_{train}), \end{aligned} \tag{6}$$

where A is a network architecture, and $\mathcal{A}$ is the search space. The goal is to search an optimal network architecture A^* that could achieve the highest generalization performance on the validation set.

Generally, search space $\mathcal{A}$ includes the following choices, (1) layer operations, e.g., perceptron, convolution, and max pooling, (2) layer hyperparameter, e.g., the number of hidden units, filters, and stride size, and (3) skip connections. We use these choices to build either the entire network architectures or cell architectures, resulting in whole-network search space and cell-based search space [35]. The former is used to construct the entire network architecture from scratch. The latter aims to design the repeated cells (normal cells and reduction cells) as shown in Figure 5 and construct the entire network architecture by stacking these cells. Due to their popularity and generality, in the rest of this section, we illustrate mainstream searching techniques that are proposed upon them.

4.1 Bayesian Optimization

When it comes to Bayesian optimization (BO) for AutoDL, there are two challenges required to overcome: (1) how to measure the similarity of two neural architectures, which are not defined in Euclidean space, and (2) how to optimize the acquisition function. The former arises from the standard BO that relies on Gaussian process as the surrogate of the objective function. Gaussian process requires measurable distance between two points to introduce a kernel function. The latter challenge appears due to discrete architectural choices in the search space, which makes it hard to optimize the acquisition function to get candidate architectures.

To resolve the first challenge, we could design a similarity measurement for two arbitrary architectures in the search space [12, 13]. Network edit-distance [12] is proposed to quantify the similarity of two networks. It calculates the number of network morphism operations [4] required to transform one network to another and introduces a valid kernel function based on the Bourgain embedding algorithm. Another similarity measurement, layer matched mass [13], measures the amount of matched computation at the layers of one network to the layers of the other in terms of the matched frequency and the location of layer operations. The corresponding distance is defined as the minimum of layer matched mass computed by an Optimal Transport (OT) program, that finds the optimal transportation plan via solving a cost minimization problem.

To address the second challenge, recent work [12] proposes A^* search with simulated annealing to optimize the acquisition function in the tree-structured search space. As network morphism [4] enables the transformation among different networks, the search space could be defined as a tree-

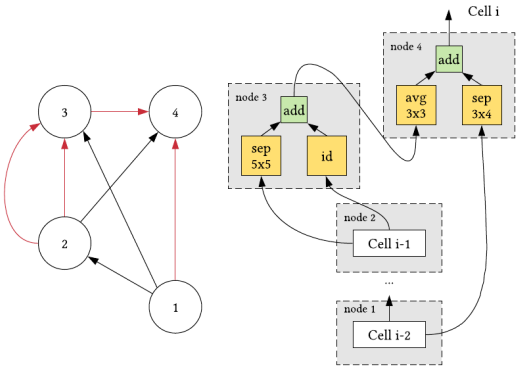

Figure 6: An example of a directed acyclic graph (DAG) for AutoDL with weight sharing [26]. The DAG is built on cell-based search space and has four nodes, where node 1 and node 2 are two previous cells' outputs, and node 3 and node 4 are blocks of the current cell. The four nodes and active edges (red) make up a sub-graph for a cell architecture. All sub-graphs share weights of layer operations used in cell neural networks.

structured graph, in which nodes are neural architectures, and edges denote morphism operations changing the architecture of the parent node. A^* search conducts exploitation by expanding the architecture of the best node in the tree while simulated annealing performs exploration, which randomly chooses other nodes for expansion with certain predefined probabilities. Evolutionary algorithm [13] is also proposed to address the challenge of searching in discrete space. It mutates N neural architectures, which have higher acquisition values by random N layer operations, evaluates the acquisition of the new ones, and repeats the above steps until the prescribed condition. Upon solving the two challenges, we could follow the standard BO procedure to solve Equation (6).

4.2 Reinforcement Learning

Within the context of AutoDL with reinforcement learning (RL), the agent is required to learn a policy to construct network architectures. The trained networks will produce validation accuracy as reward R for the agent. We explain how to train the agent in terms of two kinds of search space.

For the whole-network search space, the agent generates architectural hyperparameters of the entire neural networks. One way to train the agent is policy gradient [34]. The agent is modeled by Recurrent Neural Network (RNN) which predicts a variable string to depict the architectural hyperparameters. The policy of the RNN agent is formulated as the probability $P(a_{1:T}; \theta_c)$ of a sequence of T actions for T hyperparameters given the RNN parameters θ_c. It could be iteratively updated via policy gradient towards maximizing the expected reward $J(\theta_c) = E_{P(a_{1:T};\theta_c)}[R]$. The method [34] following REINFORCE rule expresses the reward gradient $\nabla_{\theta_c} J(\theta_c)$ as follows,

$$\nabla_{\theta_c} J(\theta_c) = \frac{1}{m} \Sigma_{k=1}^{m} \Sigma_{t=1}^{T} \nabla_{\theta_c} \ln P(a_t | a_{(t-1):1}; \theta_c) R_k, \quad (7)$$

where m is the number of sampled architectures in one batch, and R_k means the validation accuracy of neural network k. After getting the validation accuracy of a batch of architectures, we update the agent's policy by Equation (7).

For the cell-based search space, the agent only searches the architecture of normal and reduction cells. The revised RNN-agent [35] predicts a string of inputs and operations of each block within a cell. By transforming the cell descriptions into architectures and stacking them according to the base network, a complete network architecture is obtained and evaluated to provide rewards for the agent. For the agent training, another policy gradient method, Proximal Policy Optimization (PPO) [35], could be adopted.

Moreover, weight sharing [26] is an acceleration technique to search neural architectures with RL in two kinds of search space. It constructs a large directed acyclic graph (DAG) where all sub-graphs can represent all neural networks in a particular search space. Hence, it forces all child models to share parameters (weights) and avoids training each model from scratch to converge. For example, Figure 6 presents the DAG of a cell-based search space. Each node corresponds to a block of a cell, and directed edges represent how blocks are connected. A sub-graph is a cell architecture. The RNN agent is trained by policy gradient to return a string of inputs and operations for each block, that determines node computations and connected edges in the large graph. When we train a complete network using the cell description, weights of a layer operation corresponding to the node in the large graph are updated. Therefore, weights are shared with several child models.

4.3 Gradient-based Approach

For gradient-based approaches with AutoDL, the most critical challenge is to convert non-differentiable optimization problems into differentiable ones. Thus, we want to encode a network architecture into a continuous and numeric representation. Once having continuous representation, we can optimize the architectural hyperparameters with gradient descent, which is the same procedure to optimize the parameters of neural networks. Generally, there are two ways for the transformation.

One way is the mixed operation encoding [22, 2]. Given a candidate set of layer operations $\mathcal{O} = \{o_i\}$ and an input x, the output of a mixed operation $m_{\mathcal{O}}(x)$ [22] is a weighted sum of these operations.

$$m_{\mathcal{O}}(x) = \sum_{o \in \mathcal{O}} \frac{\exp(\alpha_o)}{\sum_{o^i \in \mathcal{O}} \exp(\alpha_{o^i})} o(x), \quad (8)$$

where α_o is the numerical weight for the operation o, called architectural weight. In the cell-based search space, each edge of a cell network is specified a mixed operation, and a cell with E edges is then encoded in $|\mathcal{O}| \times E$ architectural weights. Furthermore, the mixed operation can be improved its memory usage by binary gates [2], where an operation is sampled according to the probability proportional to its architectural weight. Thus, only one operation in an edge will be activated during learning rather than computing the weights of all operations. The goal of the representation is using gradient approaches to learn these architectural weights, like running 1st-order gradient descent alternatively for Equation (6). It fixes the outer architectural weights to update the gradient of inner weights and then fixes the inner weights to update the gradients of outer architectural weights. After learning, the largest value of architectural weights for an edge decides its operation.

Group	Framework	Language	Provided by	URL
Automated Feature Engineering (AutoFE)	FeatureTools [14]	Python	Open-Source	https://github.com/Featuretools/featuretools
	AutoCross [33]	-	4paradigm	https://www.4paradigm.com
Automated Model and Hyperparameter Learning (AutoMHL)	Hyperopt [1]	Python	Open-Source	https://github.com/hyperopt/hyperopt
	SMAC3 [10]	Python	Open-Source	https://github.com/automl/SMAC3
	Auto-Sklearn [9]	Python	Open-Source	https://github.com/automl/auto-sklearn
	TPOT [25]	Python	Open-Source	https://github.com/EpistasisLab/tpot
	HyperTune	-	Google	https://bit.ly/2IMsECx
	Automatic Model Tuning	-	Amazon	https://aws.amazon.com/sagemaker
	Azure AutoML	-	Microsoft	https://bit.ly/2XxBMmA
	DarwinML	-	Intelligence Qubic	http://iqubic.net/
Automated Deep Learning (AutoDL)	Auto-Keras [12]	Python	Open-Source	https://github.com/keras-team/autokeras
	AdaNet [31]	Python	Open-Source	https://github.com/tensorflow/adanet
	Google AutoML	-	Google	https://cloud.google.com/automl
	Neural Network Intelligence	Python	Microsoft	https://github.com/microsoft/nni

Table 2: A list of selective AutoML Frameworks until June 2019

Another feasible way is using encoder/decoder to learn continuous architecture representation [23]. The encoder transforms an architectural string to an embedding representation, and the decoder reverts an embedding to a string. This representation combines a performance predictor. It maps an embedding to its validation performance in the latent space where stochastic gradient descent attempts to search the embedding representation with the best performance.

5. AUTOML FRAMEWORKS

In this section, we discuss representative AutoML frameworks from either open-source project or enterprise services. Note that we exclude those open-source codes that are merely experiment implementations for research papers.

5.1 Automated Feature Engineering

FeatureTools [14] is an open-source framework using Python which can automatically generate features from relational datasets. The generation follows the relationship of any two entities to apply feature transformation operators, such as average and count. Besides, AutoCross [33] is a feature crossing tool provided by 4Paradigm. The tool automatically captures interactions between categorical features by cross-product. Its cross feature generation relies on beam search to generate candidate features. Feature selection is ranking the performance of features on a logistic regression regardless of the real learning model.

5.2 Automated Model and Hyperparameter Learning

Current AutoMHL frameworks are split into HPO and Auto-Pipeline developed from the open-source community or enterprise. We introduce them in the light of open-source and enterprise.

For open-source frameworks, Hyperopt [1] and SMAC3 [10] are tools for HPO, while Auto-Sklearn [9], TPOT [25], and H2O are frameworks for Auto-Pipeline. Hyperopt [1] and SMAC3 [10] use Bayesian optimization to tune hyperparameters of a learning model. Auto-Sklearn [9] is the extension of SMAC [10] with meta learning and ensemble learning for machine learning pipeline automation. TPOT [25] is the most popular Auto-Pipeline tool using genetic programming to optimize the tree-structure pipeline. H2O is an open-source machine learning platform written in Java developed by H2O.ai. The platform contains solutions of pipeline automation by optimizing hyperparameters via random search, grid search, and Bayesian optimization and constructing stacking ensembles.

For enterprise frameworks, Google and Amazon provide a cloud service for HPO, while Microsoft and Intelligence Qubic offer Auto-Pipeline frameworks. Google HyperTune could tune hyperparameters of machine learning models by grid and random search, TPE [1], and SMAC [10]. Amazon Automatic Model Tuning could optimize hyperparameters using Bayesian optimization. Furthermore, Microsoft Azure AutoML uses collaborative filtering and Bayesian optimization to search for promising pipelines. DarwinML is an Auto-Pipeline platform provided by Intelligence Qubic. It relies on genetic programming to construct arbitrary architectures of ensemble models for a machine learning pipeline.

5.3 Automated Deep Learning

AutoDL is a subfield of AutoMHL, but AutoDL frameworks mainly build the architectures of neural networks automatically. We introduce two well-known open-source frameworks, Auto-Keras and AdaNet, and two enterprise services, Google AutoML and Neural Network Intelligence (NNI) from Microsoft.

Auto-Keras [12] is an open-source project using Keras to search for deep network architectures. Its adopts Bayesian optimization and network morphism to search entire arbitrary neural architectures instead of cell network architectures. Moreover, AdaNet [31] is another open-source work using Tensorflow for learning network architecture automatically. It could learn network architectures and build an ensemble of networks to obtain a better model.

Besides, Neural Network Intelligence (NNI) is an open-source toolkit provided by Microsoft for neural architecture search and hyperparameter tuning. It supports a various of search techniques, e.g., evolutionary algorithm and network morphism, to tune network architectures and provides flexible execution environments in local or cloud servers. Last but not least, Google Cloud AutoML is a service to get high-quality neural networks on vision classification, text classification, and language translation. The service is based on the NAS proposed by Zoph and Le [34]. Users could obtain customized models by merely uploading their labeled datasets without any pieces of code.

6. CHALLENGES

Authoritative benchmarks are essential standard protocols for AutoML comparison that regulate datasets, the search space, and the training setting of searched machine learning solutions. Unfortunately, such benchmarks are not omnipresent in AutoML. No standard protocol for AutoFE provides a fixed set of transformation operations, and identical training setting of generated solutions. HPOLib [6] collects a set of benchmarks for HPO, but no benchmark exists for Auto-Pipeline. Researchers usually leverage the UCI Machine Learning Repository and OpenML datasets with their setting for evaluation.

For AutoDL, NAS-Bench-101 [32] is a tabular benchmark which maps five millions of trained neural architectures to their training and evaluation metrics, where these architectures are generated from the cell-based search space. As the objective comparison, any AutoDL technique generates child networks within the same search space and looks up their performance without training them from scratch, which avoids different training techniques and hyperparameters. However, the NAS-Bench-101 [32] only considers the cell-based search space. For the comparison of AutoDL within whole-network search space, one of the first attempts is Auto-Keras [12], which encompasses data preparation, training setting of child networks, and optimizer. Researchers can concentrate on inventing the AutoDL optimizer, and Auto-Keras [12] will produce the empirical performance in the standard testing environment. Authoritative benchmarks for AutoML are required; otherwise, it would be hard to compare the effectiveness and efficiency among a multitude of AutoML techniques.

Efficiency is one of the dominant factors for the successful adoption of AutoML methods. Some AutoML methods only take tolerable time to generate solutions: Auto-Sklearn [9] requires 500 seconds in a CPU to find competence pipelines; FeatureRL [18] takes 400 seconds in a CPU to find appropriate features for a learning model. In contrast, AutoDL usually demands massive computation resources and time, which could consume up to thousands of GPU days [34]. Although novel acceleration techniques are proposed to reduce the enormous order of time [12, 26], the efficiency of NAS is still an open problem.

The design of the search space is another challenge for AutoML, which requires substantial knowledge and experience from human experts. A practical search space incorporates human prior knowledge for machine learning and considers the trade-off between its size (compact) and all useful possibilities (comprehensive). A well-known example is the cell-based search [35] for AutoDL. The repeated neural network blocks in ResNet or DenseNet motivate researchers to fix the base network architecture and to search the cell typologies only. The handcrafted search spaces highly affect the performance of AutoML, which is still an intriguing question.

Interpretable analysis can provide insight on what crucial components from AutoML could lead to desired performance. With the integration of friendly user interfaces, AutoML users could also customize their solutions by selecting desirable configurations between independent runs. Nevertheless, any AutoML search techniques is still a black-box procedure. Thus, limited search knowledge from AutoML can be utilized interactively. We believe interpretability and user interaction can increase the usefulness and capability of AutoML.

Miscellaneous domains remain potential to collaborate with AutoML. Most AutoML work have focused on classical regression and classification problems, particularly AutoDL whose empirical results are majorly from image classification on CIFAR-10. Nevertheless, the optimization setup of other potential domains more or less differs from the conventional AutoML. Take anomaly detection as an example. The extremely skew distribution of the benign class against the anomaly class might lead AutoML to search biased machine learning solutions. Other domains must have distinct considerations. We believe the cooperation of AutoML with multiple potential application domains could become another promising research direction.

7. CONCLUSION

Automated machine learning (AutoML) is the end-to-end process, which automatically discovers machine learning solutions. Without laborious human involvement, AutoML can expedite the process of applying machine learning to specific domain problems. Apart from our work, there are also other AutoML reviews [7, 28], unveiling the mysterious art behind AutoML. As most of us develop with respect to three indispensable elements: search space, search techniques, and performance evaluation techniques, what makes us distinctive is that we provide a new perspective via categorizing AutoML into AutoFE, AutoMHL, and AutoDL, and elucidating them from technique dimension and framework dimension. By including and introducing both mainstream AutoML techniques and selected AutoML frameworks, we hope this survey could give insight into the progress of AutoML.

8. REFERENCES

[1] J. S. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl. Algorithms for hyper-parameter optimization. In *Advances in neural information processing systems*, pages 2546–2554, 2011.

[2] H. Cai, L. Zhu, and S. Han. ProxylessNAS: Direct neural architecture search on target task and hardware. In *International Conference on Learning Representations*, 2019.

[3] B. Chen, H. Wu, W. Mo, I. Chattopadhyay, and H. Lipson. Autostacker: A compositional evolutionary learning system. In *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO '18, pages 402–409, New York, NY, USA, 2018. ACM.

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

[5] P. M. Domingos. A few useful things to know about machine learning. *Commun. acm*, 55(10):78–87, 2012.

[6] K. Eggensperger, M. Feurer, F. Hutter, J. Bergstra, J. Snoek, H. Hoos, and K. Leyton-Brown. Towards an empirical foundation for assessing bayesian optimization of hyperparameters. In *NIPS workshop on Bayesian Optimization in Theory and Practice*, volume 10, page 3, 2013.

[7] T. Elsken, J. H. Metzen, and F. Hutter. Neural architecture search: A survey. *Journal of Machine Learning Research*, 20(55):1–21, 2019.

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

arXiv preprint arXiv:1807.01774, 2018.

[9] M. Feurer, A. Klein, K. Eggensperger, J. Springenberg, M. Blum, and F. Hutter. Efficient and robust automated machine learning. In *Advances in Neural Information Processing Systems*, pages 2962–2970, 2015.

[10] F. Hutter, H. H. Hoos, and K. Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In *International Conference on Learning and Intelligent Optimization*, pages 507–523. Springer, 2011.

[11] K. Jamieson and A. Talwalkar. Non-stochastic best arm identification and hyperparameter optimization. In *Artificial Intelligence and Statistics*, pages 240–248, 2016.

[12] H. Jin, Q. Song, and X. Hu. Auto-keras: Efficient neural architecture search with network morphism, 2018.

[13] K. Kandasamy, W. Neiswanger, J. Schneider, B. Poczos, and E. P. Xing. Neural architecture search with bayesian optimisation and optimal transport. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems 31*, pages 2016–2025. Curran Associates, Inc., 2018.

[14] J. M. Kanter and K. Veeramachaneni. Deep feature synthesis: Towards automating data science endeavors. In *2015 IEEE International Conference on Data Science and Advanced Analytics, DSAA 2015, Paris, France, October 19-21, 2015*, pages 1–10. IEEE, 2015.

[15] G. Katz, E. C. R. Shin, and D. Song. Explorekit: Automatic feature generation and selection. In *2016 IEEE 16th International Conference on Data Mining (ICDM)*, pages 979–984, Dec 2016.

[16] A. Kaul, S. Maheshwary, and V. Pudi. Autolearn: Automated feature generation and selection. In *2017 IEEE International Conference on Data Mining (ICDM)*, pages 217–226, Nov 2017.

[17] U. Khurana and H. Samulowitz. Automating predictive modeling process using reinforcement learning. *arXiv preprint arXiv:1903.00743*, 2019.

[18] U. Khurana, H. Samulowitz, and D. Turaga. Feature engineering for predictive modeling using reinforcement learning. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[19] U. Khurana, D. Turaga, H. Samulowitz, and S. Parthasrathy. Cognito: Automated feature engineering for supervised learning. In *2016 IEEE 16th International Conference on Data Mining Workshops (ICDMW)*, pages 1304–1307. IEEE, 2016.

[20] A. Klein, S. Falkner, S. Bartels, P. Hennig, and F. Hutter. Fast bayesian optimization of machine learning hyperparameters on large datasets. *arXiv preprint arXiv:1605.07079*, 2016.

[21] L. Li, K. Jamieson, G. DeSalvo, A. Rostamizadeh, and A. Talwalkar. Hyperband: Bandit-based configuration evaluation for hyperparameter optimization. 2016.

[22] H. Liu, K. Simonyan, and Y. Yang. DARTS: Differentiable architecture search. In *International Conference on Learning Representations*, 2019.

[23] R. Luo, F. Tian, T. Qin, E. Chen, and T.-Y. Liu. Neural architecture optimization. In *Advances in Neural Information Processing Systems*, pages 7826–7837, 2018.

[24] T. Nickson, M. A. Osborne, S. Reece, and S. J. Roberts. Automated machine learning on big data using stochastic algorithm tuning. *arXiv preprint arXiv:1407.7969*, 2014.

[25] R. S. Olson, N. Bartley, R. J. Urbanowicz, and J. H. Moore. Evaluation of a tree-based pipeline optimization tool for automating data science. In *Proceedings of the Genetic and Evolutionary Computation Conference 2016*, GECCO '16, pages 485–492, New York, NY, USA, 2016. ACM.

[26] H. Pham, M. Guan, B. Zoph, Q. Le, and J. Dean. Efficient neural architecture search via parameters sharing. In J. Dy and A. Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 4095–4104, StockholmsmÃďssan, Stockholm Sweden, 10–15 Jul 2018. PMLR.

[27] F. Qi, Z. Xia, G. Tang, H. Yang, Y. Song, G. Qian, X. An, C. Lin, and G. Shi. Darwinml: A graph-based evolutionary algorithm for automated machine learning. *arXiv preprint arXiv:1901.08013*, 2018.

[28] Y. Quanming, W. Mengshuo, J. E. Hugo, G. Isabelle, H. Yi-Qi, L. Yu-Feng, T. Wei-Wei, Y. Qiang, and Y. Yang. Taking human out of learning applications: A survey on automated machine learning. *arXiv preprint arXiv:1810.13306*, 2018.

[29] K. Swersky, J. Snoek, and R. P. Adams. Freeze-thaw bayesian optimization. *arXiv preprint arXiv:1406.3896*, 2014.

[30] B. Tran, B. Xue, and M. Zhang. Genetic programming for feature construction and selection in classification on high-dimensional data. *Memetic Computing*, 8(1):3–15, 2016.

[31] C. Weill, J. Gonzalvo, V. Kuznetsov, S. Yang, S. Yak, H. Mazzawi, E. Hotaj, G. Jerfel, V. Macko, B. Adlam, M. Mohri, and C. Cortes. Adanet: A scalable and flexible framework for automatically learning ensembles, 2019.

[32] C. Ying, A. Klein, E. Real, E. Christiansen, K. Murphy, and F. Hutter. Nas-bench-101: Towards reproducible neural architecture search. *arXiv preprint arXiv:1902.09635*, 2019.

[33] L. Yuanfei, W. Mengshuo, Z. Hao, Y. Quanming, T. WeiWei, C. Yuqiang, Y. Qiang, and D. Wenyuan. Autocross: Automatic feature crossing for tabular data in real-world applications. *arXiv preprint arXiv:1904.12857*, 2019.

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

[35] B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le. Learning transferable architectures for scalable image recognition. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2018.