Techniques for Automated Machine Learning

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

ABSTRACT

Automated machine learning (AutoML) aims to find optimal machine learning solutions automatically given a problem description, its task type, and datasets. 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 portray AutoML as a bi-level optimization problem, where one problem is nested within another to search the optimum in the search space, and review the current developments of AutoML in terms of three categories, automated feature engineering (AutoFE), automated model and hyperparameter tuning (AutoMHT), and automated deep learning (AutoDL). State-of-the-art techniques in the three categories are presented. The iterative solver is proposed to generalize AutoML techniques. We summarize popular AutoML frameworks and conclude with current open challenges of AutoML.

1. INTRODUCTION

Machine learning has been broadly implemented in a myriad of fields, from image classification [36], speech recognition [35], and recommendation platforms [17], to beating human champion in Go games [78]. Automated machine learning (AutoML) has emerged as a prevailing research upon the ubiquitous adoption of machine learning. It aims at determining high-performance machine learning solutions with a little workforce in reasonable time budget. For example, Google HyperTune [34], Amazon Model Tuning [1], and Microsoft Azure AutoML [65] all provide cloud services cultivating off-the-shelf machine learning solutions for both researchers and practitioners. Therefore, AutoML brings about three advantages: (1) liberating the community from the time-consuming and trial-and-error tuning processes, (2) facilitating the development of machine learning in organizations or business, and (3) making machine learning universally accessible to all domain scientists.

The canonical 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 classify AutoML into three categories (Figure 2): (1) automated feature engineering (AutoFE), (2) automated model and hyperparameter tuning (AutoMHT), and (3) automated deep learning (AutoDL). AutoFE discovers informative and distinct features for the learning model. AutoMHT tunes hyperparameter of a specific learning model (a.k.a. HPO [26]) or an entire machine learning pipeline automatically (Auto-Pipeline). AutoDL is a subfield of AutoMHT focusing on the automatic design of neural networks. Since deep learning has succeeded in image and language tasks to extract hierarchical features in an end-to-end manner without domain understanding, we explicitly separate AutoDL from AutoMHT.

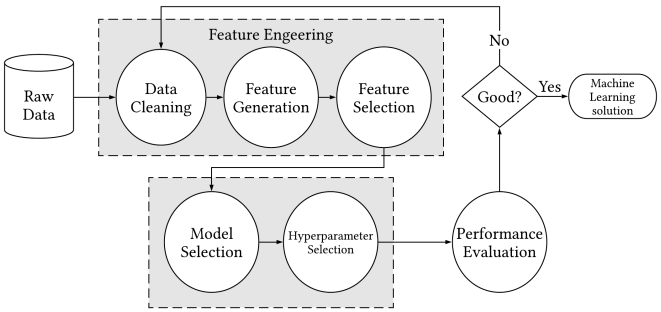

Figure 1: A classical machine learning pipeline. When data scientists intend to find their machine learning solutions, they manually tune the combination of features, models, and hyperparameters in a pipeline until the final solution achieves low generalization error. AutoML is proposed to facilitate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

Like machine learning, the essence of AutoML is an optimization problem, specifically bi-level optimization [16], with an expansive search space including features, hyperparameters, models, and network architectures. AutoML is an NP-complete problem as intractable as machine learning. As a result, several heuristic techniques are proposed to solve the sophisticated optimization. On top of the categorization of AutoML, we will review three categories of AutoML in terms of techniques and frameworks. The techniques span optimization approaches, including reinforcement learning, evolutionary algorithm, Bayesian optimization, and gradient approaches. We generalize them by the iterative solver (Figure 3) and explain how these techniques are deployed in AutoML. In the framework dimension, we list representative AutoML frameworks in commercial services and open source communities to give AutoML beginners painless entries.

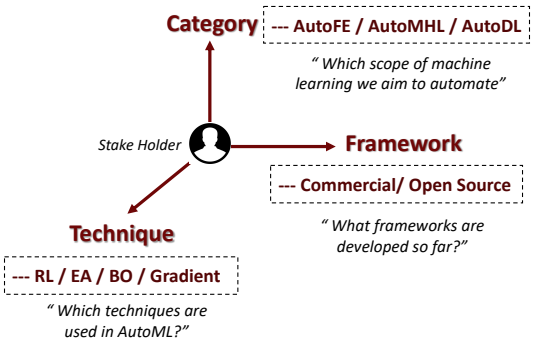

Figure 2: A taxonomy of AutoML. We split AutoML into automated feature engineering (AutoFE), automated model and hyperparameter tuning (AutoMHT), and automated deep learning (AutoDL). The AutoML techniques include Bayesian optimization (BO), reinforcement learning (RL), evolutionary algorithm (EA), and gradient approaches (Gradient). We also review AutoML frameworks in commercial services and open source communities.

Several surveys on AutoMHT [101], AutoDL [24; 94; 37], and general AutoML [72] have been published from different perspectives, unveiling the mysterious art behind AutoML. One can sum up AutoML via the setup of reinforcement learning [72], while the dominant anatomy of AutoML is search space, search techniques, and performance evaluation proposed in [24]. The comprehensive analysis for the search space and techniques of AutoDL could be found in [94], including constrained and multi-objective optimization. The techniques of data augmentation for AutoDL are well organized in [37]. For the detailed techniques of AutoMHT and Auto-Pipeline, one could be found in [101] with performance analysis over real datasets.

Different from existing surveys, we map AutoML to the optimization paradigm and elucidate AutoML techniques from the iterative solver (Figure 3). In the manner, the optimal solution is produced by the iterative procedure, where the explorer seeks out candidates and the evaluator inspects the effectiveness of the candidates. This illustration bridges the bi-level optimization [16] and AutoML. The contributions of this paper are in the following,

- We illustrate AutoML from the formal statement of machine learning, depict AutoML as a bi-level optimization problem, and display their optimization formats for the AutoFE, AutoMHT, and AutoDL.
- We propose the iterative solver to generalize AutoML techniques and analyze these techniques commonly used in AutoML. Readers will learn how to apply them to three categories in AutoML.
- We review the commercial and open-source frameworks in AutoML. Readers can get available AutoML tools for their research.

2. PRELIMINARIES

Before diving into the techniques of automated machine learning (AutoML), we examine the essence of AutoML by

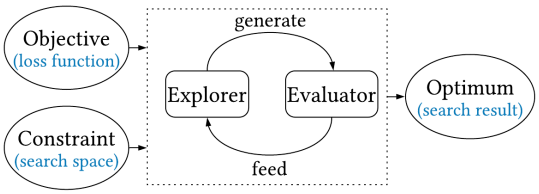

Figure 3: The iterative solver (dotted square block) generalizes AutoML techniques. In the optimization field, the optimum of an objective is solved under the constraint. We map AutoML to the paradigm. The loss function stands for the objective, and the search space represents the constraint. AutoML techniques solve the optimization in an iterative manner, where the explorer generates an possible solution, and the evaluator feeds back the solution performance to the explorer.

re-visiting the formal problem statement of machine learning and point out the discrepancy between them.

Machine learning is to determine a prediction function A such that $A(x)$ can accurately predict the true output y. Without loss of generality, $A(x;w) : \mathcal{R}^{d_x} \times \mathcal{R}^d \to \mathcal{R}^{d_y}$, is parameterized by w, and belongs to $\mathcal{A} := \{A(.;w) : w \in \mathcal{R}^d\}$. We aims to get the optimal A^* in $\mathcal{A}$, i.e., getting the optimal w^* for A^*. A given loss function $\mathcal{L} : \mathcal{R}^{d_y} \times \mathcal{R}^{d_y} \to \mathcal{R}$ can quantify the inaccurate prediction. The optimization statement is as follows,

$$\arg\min_{A\in\mathcal{A}} R_n(A) = \arg\min_{w\in\mathcal{R}^d} R_n(w) = \sum_{i=1}^{n} \mathcal{L}(A(x_i;w), y_i), \quad (1)$$

where $R_n(.)$ is empirical risk. Actually, we are supposed to minimize expected risk $R(w) = \mathbb{E}[\mathcal{L}(A(w;x), y))]$, but since we would obtain partial samples from the data distribution in practice, we could only minimize $R_n(.)$ instead. To avoid the gap between empirical risk and expected risk, the structural risk minimization [86] is proposed to split the available dataset to training set, validation set, and testing set, which is the common technique among machine learning practitioners. The training set could train a set of candidates with different w values. The validation set could estimate the empirical risk among the candidates to choose the one yielding the lowest risk. The testing could measure the expected risk for the chosen candidate.

In context of AutoML, there are two differences from machine learning. First, we relax $\mathcal{A}$ to $\hat{\mathcal{A}}$. Unlike machine learning whose search space $\mathcal{A}$ consists of single-type candidates parameterized by w, $\hat{\mathcal{A}}$ could encompass mixed model types, such as logistic regression, support vector machines, and decision tree at the same time. $\hat{\mathcal{A}}$ could also be a pool of diverse neural network architectures. Second, AutoML is formulated by bi-level optimization. AutoML wants to discover the best one (1st-level optimization) among several well-trained machine learning solutions (2nd-level optimization). Mathematically, AutoML is presented in Equation (2).

$$\begin{aligned} A^* &= \arg\min_{A\in\hat{\mathcal{A}}} R_n(A(x_{val}; w^*)) \\ \text{s.t.} \quad w^* &= \arg\min_{w} R_n(A(x_{train}; w)). \end{aligned} \quad (2)$$

Note that w symbolizes trainable parameters of A, and dif-

Category \ Technique	Bayesian Optimization	Reinforcement Learning	Evolutionary Algorithm	Gradient Approach	Otherss
Automated Feature Engineering (AutoFE)	-	FeatureRL [49] NFS [14]	Genetic Programming Feature Engineering [84] Genetic Programming Skewed Feature Selection [88]	-	Data Science Machine [45] ExploreKit [46] Cognito [50] AutoLearn [47] OneBM [54] LFE [67]
Automated Model and Hyperparameter Tuning (AutoMHT)	TPE [6] SMAC [39] Auto-Sklearn [27] FABOLAS [51] BOHB [25]	APRL [48] Hyperband [56]	TPOT [69] Autostacker [11] DarwinML [71]	-	-
Automated Deep Learning (AutoDL)	AutoKeras [42] NASBot [44]	NAS [102] NASNet [103] ENAS [70]	LargeEvoNet [74] AgingEvoNet [73] HierEvoNet [59] MorphEvoNet [93]	DARTS [60] Proxyless [10] NAONet [61] NASP [95]	-

Table 1: An overview of different techniques for automated machine learning.

ferent models do not necessarily have the same format of w. Hence, AutoML considers all tunable arguments into the whole optimization procedure. It automates the manual tuning procedure giving us the appearance of "automation." All in all, AutoML is an optimization problem with flexible search space and pushes machine learning to examine a broad scope of candidate solutions.

Despite the automation advantage, AutoML is still NP-complete. Specifically, feature engineering and hyperparameter tuning are applications of combinatorial optimization [52], that searches optimum of an objective function over a discrete but gigantic configuration space. Its complexity has been proven as NP-complete [52]. Moreover, training a 3-layer neural network to get optimal weights is NP-complete [7]. It would be straightforward to derive that the AutoDL is also NP-Complete by the following two statements. (1) Any optimal neural network can be verified in polynomial time. (2) If the number of network candidates is one, searching network architectures is equivalent to training the neural network. Thus, we can reduce the training problem to the search problem. The NP-complete complexity of AutoML implies even though you could have found the best model, you are wondering if a better model would pop up if you keep searching.

Fortunately, heuristic techniques are proposed to locate the local optimal as close as possible to the global one. They move toward the optimal in an iterative manner. We generalize these AutoML techniques by the iterative solver as characterized in Figure 3. Given a search space, an explorer picks a candidate solution for an evaluator. The evaluator assesses the solution to give the explorer a feedback. For example, the acquisition function in BO is the explorer that selects different hyperparameters or features for machine learning models. The cross validation accuracy is the evaluator's feedback to update the surrogate model of the explorer.

The remainder of the paper is organized as follows: Section 3 discusses AutoFE. Section 4 explains AutoMHT. Section 5 illustrates AutoDL. Section 6 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 1 summarizes the AutoML techniques mentioned in the work.

3. AUTOMATED FEATURE ENGINEERING

Feature engineering manipulates features via data imputation, feature generation, feature selection, and so on. The dataset $\mathcal{D}^F$ has its feature set F. Applying pre-processing operations $\mathcal{T}$ to $\mathcal{D}^F$ generates the derived dataset $\mathcal{D}^{\hat{F}}$. The automated feature engineering could be formulated in the following optimization problem,

$$\begin{aligned} F^* &= \arg\min_{F \in \mathcal{F} \cup \hat{\mathcal{F}}} \mathcal{L}_{val}(A(D^F_{val}; w^*)), \\ \text{s.t.} \quad w^* &= \arg\min_{w} \mathcal{L}_{train}(A(D^F_{train}; w)), \end{aligned} \tag{3}$$

where $\mathcal{L}_{(.)}$ is the loss function for training and validation, and $D^F_{(.)}$ denotes training and validation set from D^F. AutoFE intends to pick up vigorous features that can boost the performance of algorithm A through feature generation and feature selection. Feature generation can augment the original features by pre-preocessing methods, e.g., normalization. Feature selection will reduce feature dimensions by principal component analysis (PCA) [19] and linear discriminant analysis (LDA) [77], or filter overlapping features by information gain [75]. In this section, we briefly introduce reinforcement learning (RL) and evolutionary algorithm (EA), and explain how RL, EA, and other techniques are used in AutoFE.

3.1 Reinforcement Learning

Reinforcement learning (RL) [81] 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 policy that defines the agent's strategy could be a transformation graph (Figure 4) or recurrent network network (Figure 5). The agent generates potential feature transformations or different hyper-parameters as its actions. The environment evaluates machine learning models using those actions to feed the validation performance back to the agent. In the context of the iterative solver, the agent and the environment correspond to the explorer the evaluator, respectively. RL repeatedly generates actions and evaluates performance until it discovers competent features or hyperparameters for models.

For AutoFE, a transformation graph [49] contains all combinations of features and transformation functions, in which a traversal from the root to a leaf node is a feature engineering (Figure 4). The root node is the original dataset. Edges are

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
```

feature transformations and feature selection. Other nodes are the derived features. The goal is to learn the traversal to reach the optimal node bringing about the highest performance. Another AutoFE representation is constructing an agent composed of n recurrent neural networks (RNNs) for n features [14]. Each RNN (Figure 5) will predict a series of T transformations for one feature. In other words, $n \times T$ feature transformations are the actions. The goal of this representation is to obtain proper $n \times T$ transformations augmenting the original features.

We could train agents by Q-Learning [49] and policy gradient [14]. First, Q-Learning [49] could help the agent learn the exploration in the transformation graph. The action (n, t) is a pair of an existing node n and the feature transformation t to n. The ϵ-greedy algorithm determines the next action: (n, t) is chosen randomly with probability ϵ or from the maximum of Q function with probability $1 - \epsilon$. The validation performance with augmented features serves as the reward r. To simplify the explosive feature combinations, approximate Q function [49] is proposed in $Q(s, a) = w^a f(s)$, where w^a is a weight vector, and $f(s)$ is a state vector. The weight vector is updated with the learning rate α and the discount factor γ by Equation (4).

$$w^{a_t} \leftarrow w^{a_t} + \alpha \times f^{(t)}(s) \times (r^{(t)} + \gamma \max_{a'}[Q(s^{(t+1)}, a') - Q(s^{(t)}, a)]). \quad (4)$$

Next, the agent can learn their parameters θ_c of the RNN controller [14] by policy gradient [82] and REINFORCE rule [92]. Its objective is to maximize the expected return of $n \times T$ transformations, $J(\theta_c) = E_{P(a_{1:n\times T};\theta_c)}[\Sigma_{t=1}^{n\times T} R_t]$, where $P(a; \theta_c)$ is the probability of a transformation, and R_t is the performance gain of a transformation. θ_c could be iteratively updated via the following gradient

$$\nabla_{\theta_c} J(\theta_c) = \frac{1}{m} \Sigma_{k=1}^{m} \Sigma_{t=1}^{n\times T} \nabla_{\theta_c} \ln P(a_t | a_{(t-1):1}; \theta_c) R_t[k], \quad (5)$$

where m is the number of different augmented feature sets, and $R_t[k]$ means the cross-validation score of the k-th set.

3.2 Evolutionary Algorithm

Evolutionary algorithm (EA) [99] finds the optimal solution from the mutation of random initial individuals, just like the natural evolution. Typically, EA follows Algorithm 2, including four phases, population, selection, crossover, and mutation. In the beginning, EA randomly generates individuals as the initial population. Individuals could be machine learning models with different hyperparameters or features. We measure the utility of the population by performance metrics, such as validation accuracy. Selection methods (e.g., tournament selection [31]) will pick up competent individuals as parents. Mutation/crossover will change the parents' structures to create various individuals. For example, using different feature transformations or changing hyperparameter values are mutation/crossover to form the next population. The above iteration will be repeated until we find qualified solutions. In the context of the iterative solver, the mutation/crossover maps to the explorer and the population measurement maps to the evaluator.

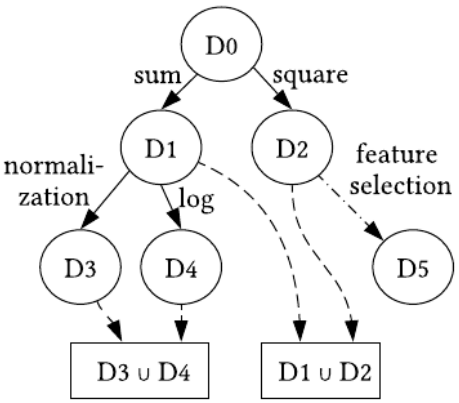

Figure 4: The transformation graph [49] for AutoFE. D_i is a dataset with different features, while edges are feature engineering operations, such as sum, square, log, normalization, and feature selection. A traversal from the root to a leaf node is a way to generate new features for the initial dataset.

Equipped AutoFE with EA, derived features for individuals could be represented in the tree structure [84] (Figure 6), where the root and the intermediate nodes are arithmetic operators, and leaf nodes are either original features or random values. From the lowest level of the tree, we can apply arithmetic operators to the leaf nodes to generate new features. Besides, we can generate another derived features with different leaf and intermediate nodes [88]. Leaf nodes are subsets of features selected by selection metrics, such as information gain [75], odds-ratio [66], and correlation coefficient [75]. Intermediate nodes become set operations (intersection, union, and difference) to combine select features.

Genetic programming [84; 88] will learn the optimal features by modifying the intermediate nodes and leaves of individuals in the mutation and crossover phase. Specifically, crossover would randomly exchange parts of two selected individuals, while mutation would randomly alter part of an individuals. The selection phase picks up individuals based on cross-validation score for the next population.

3.3 Other Methods

Besides RL and EA, there are other approaches [45; 46; 50; 47; 54; 67] 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.

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**
    $\mathcal{M}(i)$; `// 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) $\cup$ Mutation(G_2)
end
return The individual with the highest fitness

For feature generation, we can categorize datasets to relational and non-relational. Relational datasets, depicted by an entity graph, provide relationships between entities. Following the relationships, we make use of arithmetic or aggregate operators to generate additional features [45; 54]. For example, given a max depth, all features are collected by traversing all possible paths in the entity graph [54], whose depth is not greater than the max one. Another example is looking forward and backward entities of the target entity to collect features [45]. Compared with the first one [54], the second one [45] focuses on a narrow scope and produces compact derived features by transformation functions. Regarding 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 [47]. If datasets are expressed in the transformation graph (Figure 4), depth-first and breath-first traversals [50] without training an agent offer a simple way to decide the next node and feature transformation. In addition, the effective feature generation could be learned from the pattern of previous datasets. These approaches [46; 67] require meta features of different datasets to train a meta classifier (random forest [46] or neural network [67]). The meta classifier would predict hypothetical transformations for a new dataset.

For feature selection, we can evaluate features by Chi-square hypothesis [55; 67], a tree model [47; 46] or a machine learning pipeline [45]. Chi-square hypothesis [55] could test whether a dependency exists between features and the targets, which provides a simple and fast way to determine potential features without training predictive models [67]. Also, one could select final features [47] by training a decision tree to get the information gain and aggregate many selection algorithms on different subsets of features. Another way is constructing a random forest (RF) model over meta-features between datasets [46]. The rank of new features from the RF model is used to select features. Finally, one could build a machine learning pipeline with Truncated SVD for feature selection [45]. The SVD first reduces generated features. Then we keep top $\gamma\%$ components of the SVD according to their variances w.r.t the target values (F-test value). The percentage of kept features is determined by the pipeline performance.

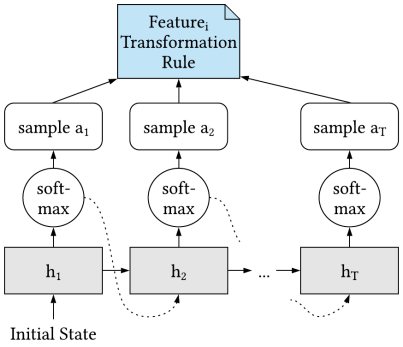

Figure 5: The recurrent neural network (RNN) generates the transformation rule for i-th feature. The n RNNs for n features forms the agent in [14]. The transformation rule is made up of a series of transformations $[a_1, a_2, ..., a_T]$, where each a_t is sampled beyond the output of softmax classifier of hidden layer h_t.

3.4 Discussion

Feature engineering is an inevitable application in commercial companies, especially e-commence, banks, and social media, because enormous and valuable data are stored in relational databases. Currently, the exploration-reduction techniques (Section 3.3) are the most popular AutoFE, among which Deep Feature Synthesis (DFS) [45] has caught the most attention with its commercial service, FeatureTool. OneBM [54] defeats DFS [45] in Kaggle competitions since it traverses comprehensive features for feature generation. Both DFS [45] and OneBM [54] are able to extract distinguishing features from relational databases. Nevertheless, the efficiency of AutoFE is far from satisfactory. One might integrate domain knowledge or leverage the mutual information from information theory [87] to discern useful features.

4. AUTOMATED MODEL AND HYPERPARAMETER TUNING

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 and a measurement $\mathcal{L}_{val}$ for validation. Considering a set of learning models $\mathcal{A} = \{A^{(1)}, A^{(2)}, ...\}$, automated model hyperparameter turning (AutoMHT) could be formulated as the following optimization problem,

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

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 $\mathcal{A} = \{A^{(1)}\}$, AutoMHT becomes hyperparameter optimization (HPO). If we expand $\mathcal{A}$ to include features F_i, i.e., $\mathcal{A} = \{A^{(1)}, A^{(2)}, ...F_1, F_2, ...\}$, AutoMHT becomes the automated pipeline learning (Auto-Pipeline). Therefore, AutoMHT is the general form of the above two AutoML problems. In this section, we briefly introduce BO and review AutoMHT in terms of BO, RL, and EA.

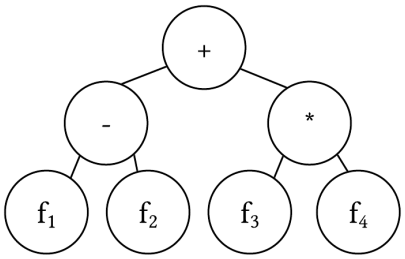

Figure 6: A feature tree used in genetic programming [84]. The leaf nodes are original feature values or random constant values. The root and intermediate nodes are arithmetic operations. The derived features are generated by $(f_1 - f_2) + (f_3 \times f_4)$.

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 $\mathcal{D}_n$

4.1 Bayesian Optimization

Bayesian optimization (BO) [76] is an iterative procedure in Algorithm 3 to finds the optimum with the two indispensable components, a probabilistic model and an acquisition function. The probabilistic model is a surrogate emulating the objective function in the search space$f$ by Gaussian processes [79] or random forest regressions [39]. For example, Gaussian processes can capture the relationship between validation accuracy and SVMs using different hyperparameters. The acquisition function, e.g., upper confident bound and expected improvement, estimates the utility of points by considering the trade-off between exploration (trying unseen points) and exploitation (improving current good points). Optimizing the acquisition brings about the potential points (e.g., new hyperparameters for SVMs) close to the optimal solutions. In the context of the iterative solver, the acquisition function using the probabilistic model serves as the explorer, and evaluating model performance serves as the evaluator.

There are two key challenges to be considered when applying BO for AutoMHT, 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 SVMs as an example. The kernel is a categorical hyperparameter having the choices of linear, polynomial, and RBF. The degree is a discrete hyperparameter specifying the degree of polynomial kernel function while the gamma is a continuous hyperparameter controlling the kernel coefficient for RBF. The efficiency bottleneck usually arises from the performance evaluation step. We need to obtain the real value of the objective function every iteration. The situation becomes worse when it is costly to train learning models (e.g., neural networks) on a large-scale dataset. Therefore, various techniques have been proposed to address these challenges.

4.1.1 Heterogeneous Search Space

Tree-based BO [39; 6] has emerged as alternates to handle the heterogeneous search space. Gaussian processes in standard BO cannot consider the hierarchical dependency and cannot model the probabilities of various types of hyperparameters. Random forest (RF) regression could replace it as the surrogate model [39]. Random subsets of historical observations build individual decision trees. A tree node decides the value to split a discrete/continuous hyperparameter or whether a hyperparameter is active to handle conditional variables. An acquisition function needs the probability $p(c|\lambda)$ of real performance c given hyperparameters λ. RF regression does not produce $p(c|\lambda)$, but the frequency estimates of μ_λ and σ_λ^2 from individual trees could be used to approximate $p(c|\lambda) = \mathcal{N}(\mu_\lambda, \sigma_\lambda^2)$.

Another way to replace Gaussian processes is building a probability density tree [6], in which a tree node stands for the probability density of a specific hyperparameter rather than a decision. There are two types of probability density in a node,

$$p(\lambda|c) = \begin{cases} l(\lambda) = p(\lambda|c < c^*) \\ g(\lambda) = p(\lambda|c >= c^*). \end{cases} \tag{7}$$

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. [6] 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 [27] by adding pipeline operations, such as data imputation, feature scaling, transformation, and selection. The tree-based BO [39; 6] could find the optimal pipeline solution in the new search space. However, the new space is much larger than that of hyperparameter tuning. Considering past performance on similar datasets [27] enables BO to start in the small but reasonable regions, which reduces the number of intermediate results. For those sub-optimal pipelines, we can also build a weighted ensemble of them [27] to get robust performance.

4.1.2 Approximate Performance Evaluation

Approximate performance evaluation, also called multi-fidelity optimization, is proposed to decrease the evaluation cost. Specifically, we could train the model in an economical setting. For example, we can use a subset of training data [68] or fewer epochs [83] of stochastic gradient descent 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, (1) the loss function for performance evaluation and (2) the training time function, both of which

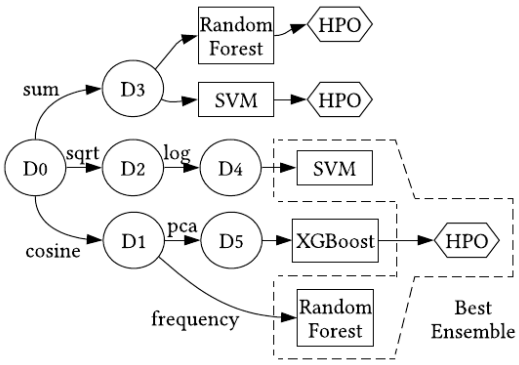

Figure 7: The exploration tree [48] for AutoMHT with RL. The root D_0 is the initial dataset, and circular nodes are derived datasets by applying feature transformations. Rectangular nodes stand for machine learning algorithms. HPO indicates whether we tune hyperparameters of the learning model. The goal is to find the ensemble of learning models for a machine learning pipeline.

consider the size of the training set as an additional input [51]. Bayesian optimizer could determine the proper size of training data that will take less training time and ideally represent the performance of the full training set. 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 [56] is a representative example illustrated in Section 4.2. An incremental work BOHB [25] using Hyperband to speed up a tree-based BO approach [6]. Different from Hyperband sampling configurations by random search, BOHB relies on a model-based approach (TPE) [6] to select potential configurations, making it easier to converge the performance of resulting configurations.

4.2 Reinforcement Learning

Speaking of AutoMHT with RL, the agent could determine the hyperparameters of a learning algorithm [56] or design a machine learning pipeline (Auto-Pipeline) [48].

For the hyperparameter tuning, we could frame it in the multi-armed bandit, a particular case of RL, where 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 [56]. 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 [41] assigns each configuration $\frac{B}{n}$ budgets and then selects 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 [56]; large n configurations occupy few budgets while few n configurations consume large budgets. The two-fold dynamic resource allocation reduces the evaluation time [56]. Thus, when n arms are sampled by random search [56] or tree-based BO [25], the aforementioned multi-armed techniques could efficiently determine the best one.

For Auto-Pipeline, we could picture its search space to the exploration tree [48] (Figure 7), 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 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 next operation. Following the Equation (4), the Q function for actions is learned.

4.3 Evolutionary Algorithm

Extant AutoMHT techniques with EA [69; 11; 71] focus on automated pipeline learning (Auto-Pipeline). Genetic programming particularly encodes pipelines in flexible structures and evolves individuals into the one with the highest predictive performance. There are three types of flexible pipeline structures, tree-based [69], layer-based [11], and graph-based pipeline [71].

First, in a tree-based pipeline [69], leaves are the copies of input data. Intermediate nodes are pipeline operators, such as feature pre-processing methods, or feature selection algorithms. The root node must be a learning model. The tree pipeline has one or more pre-processing nodes followed by a selection node and a model node. The mutation and crossover of genetic programming might change the types of pipeline operators as well as hyperparameters of each pipeline operators. The individual fitness is the predictive performance of the model node using the processed features of intermediate nodes.

Second, a layer-based pipeline [11] is similar to a multiple layer perceptron, but each neuron is a machine learning model. The first layer is the input data. The outputs 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. Genetic programming tunes the above space to find the optimal pipeline.

Finally, the most general pipeline structure is a directed acyclic graph (DAG) [71], 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 adjacency matrix controls the edge connections of a DAG, which is evolved by mutation operators. After genetic programming designs the pipeline structure and learning models, associate hyperparameters could be tuned by Bayesian optimization. Such pipeline structure does not explicitly operate feature engineering, but it leverages outputs of learning models as new features.

4.4 Discussion

AutoMHT is a historic research topic in the machine learning community from model selction, hyperparameter optimization, to pipeline optimization. Before the popularity of

AutoML, random search [5] and grid search have been the default choices for practitioners to tune hyperparameters. According to the benchmark [101], Hyperopt using TPE [6] outperforms SMAC [39], BOHB [25], FABOLAS [51], even random and grid search. Moreover, Hyperband [56] speeding up the evaluation by successive halving attains better performance [56] than Hyperopt. To our best experience, Hyperopt and Hyperband could serve as baselines for hyperparameter tuning. When it comes to search machine learning pipelines, we recommend both Auto-Sklearn [27] and TPOT [69] because they include a large number of feature engineering and models. We can easily obtain well-performance pipelines as long as we get datasets ready. In fact, the pipeline layout also matters the performance. Nonetheless, there is no mature techniques to optimize pipeline structures. We believe that apart from the innovative techniques of hyperparameter tuning and quick performance evaluation, the pipeline architecture search is still a promising direction.

5. 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 AutoMHT. Following Equation (6), we can describe AutoDL in the following optimization problem,

$$\begin{aligned} A^* &= \arg\min_{A \in \mathcal{A}} \mathcal{L}_{val}(A(D_{val}; w^*)), \\ \text{s.t.} \quad w^* &= \arg\min_{w} \mathcal{L}_{train}(A(D_{train}; w)), \end{aligned} \tag{8}$$

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. Note that Equation (8) is mainly for the optimization of neural architectures, in which other non-architecture hyperparameters, like learning rates, momentum, weight decay, are not considered.

Generally speaking, 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 [103], respectively. 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 8 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 search techniques that are proposed upon them. Non-architecture hyperparameters tunining in AutoDL is not discussed.

5.1 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 is not defined in Euclidean space, and (2) how to optimize the acquisition function. The former arises from the standard BO that relies on Gaussian processes as the surrogate model. Gaussian processes require measurable distances between two points. 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.

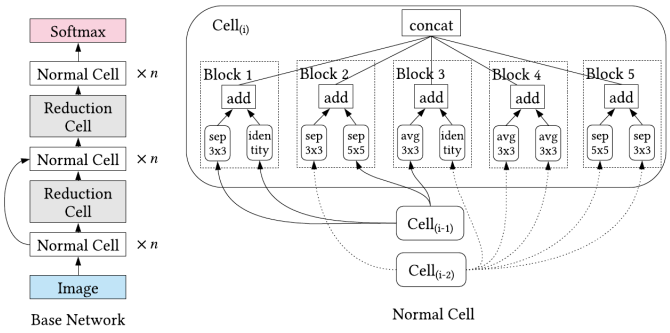

Figure 8: An example of cell-based search space [103]. The left figure is a base network that stacks normal cells and reduction cells with skip connections. The right one is a normal cell, a convolutional cell that returns a feature map of the same dimension. A reduction cell is also a convolutional 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. 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.

To resolve the first challenge, we could design a similarity measurement for two arbitrary architectures in the search space [42; 44]. Network edit-distance [42] is proposed to quantify the similarity of two networks. It calculates the number of network morphism operations [12] required to transform one network to another and introduces a valid kernel function based on the Bourgain embedding algorithm [9]. Another similarity measurement, layer matched mass [44], measures the amount of matched computation 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 [89], that finds the optimal transportation plan via solving a cost minimization problem.

To address the second challenge, recent work [42] proposes A^* search with simulated annealing to optimize the acquisition function in the tree-structured search space. As network morphism [12] enables the transformation among different networks, we could define the search space in the tree-structured graph, where nodes are neural architectures, and edges denote morphism operations changing the architecture of the parent node. A^* search exploits architectures by expanding the best node in the tree, while the simulated annealing explores architectures by randomly expanding other nodes with certain predefined probabilities. Evolutionary algorithm [44] is also proposed to address the challenge of the discrete space. It mutates the layer operations of N neural architectures and then evaluates the utilities of the mutated ones until the stop condition. Upon solving the two challenges, we could follow the standard BO procedure to solve Equation (8).

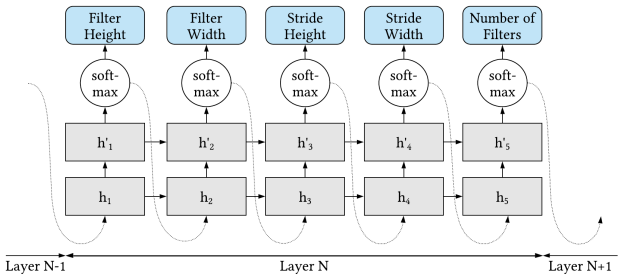

Figure 9: The RNN controller [102] predicts a sequence of strings for whole network. Each step of RNN outputs a layer hyperparameter (e.g., filter height and width) for a neural networks that would be evaluated. The RNN controller repeatedly predicts until all layers of the whole network get their hyperparameters.

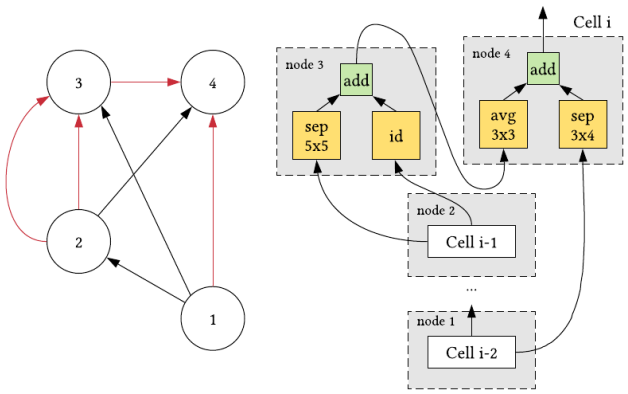

Figure 10: An example of a directed acyclic graph (DAG) for AutoDL with weight sharing [70]. The DAG is built on cell-based search space with four nodes, where node 1 and node 2 are two previous cells' outputs, and node 3 and node 4 are blocks of the current cell. The four nodes and active edges (red) make up a sub-graph for a cell architecture. All sub-graphs share weights of networks operations (e.g., sep, dilated, or conv) used in cell neural networks.

5.2 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 [102]. The agent is modeled by Recurrent Neural Network (RNN), as presented in Figure 9, 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 [102] following REINFORCE rule expresses the reward gradient $\nabla_{\theta_c} J(\theta_c)$ as follows,

$$\nabla_{\theta_c} J(\theta_c) = \frac{1}{m} \Sigma_{k=1}^{m} \Sigma_{t=1}^{T} \nabla_{\theta_c} \ln P(a_t | a_{(t-1):1}; \theta_c) R_k, \quad (9)$$

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 policy by Equation (9).

For the cell-based search space, the agent only searches the architecture of normal and reduction cells. The revised RNN-agent [103] predicts a string of inputs and operations of each block within a cell. By transforming the cell descriptions into architectures and stacking them, 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) [103], could be adopted.

Moreover, weight sharing [70] is an acceleration technique to search neural architectures. It constructs a large directed acyclic graph (DAG) where all sub-graphs can represent all cell 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 10 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. 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.

5.3 Evolutionary Algorithm

When researchers apply evolutionary algorithm (EA) to AutoDL [74; 73; 59; 93], they define a network architecture as an entity. The fitness of an entity is the accuracy on the validation set if the task is an image classification. Parents are picked up from the previous population (generation) based on fitness by tournament selection [31]. Mutation operations change the parent architectures to generate diverse child networks. The parent selection is analogous to an architecture exploitation and child generation is an architecture exploration. After several generations, EA can discover networks with good performance.

EA-based AutoDL varies in terms of search space, selection methods, and acceleration. One could search an entire network architecture from scratch [74], while others [73; 59; 93] could search cell architectures, which stacks together to form a complete deep network. Regarding the cell-based search space, one [73] could follow the structure of NASNet [103] or hierarchical cell architectures [59], which enables EA to learn the structure of the base network and cell architectures in the DAG simultaneously. Regardless of types of search spaces, mutation could change layer operations in whole network architecture or in cells, and determine which inputs are used to the next cell/layer.

For the selection methods, one could pick up the highest fitness among two random entities [74], that is a simplified tournament selection [31]. Or we could add the age information, that play a role of regularization, in the old generation to keep younger entities in a new generation [73]. For the acceleration, child networks could inherit the weights of par-

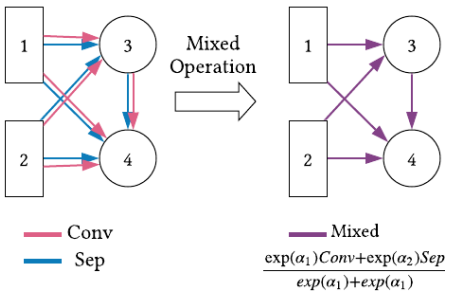

Figure 11: The continuous representation using the mixed operation [60; 10; 95] for a 4-node cell network. The first two nodes (square) are two previous cells' outputs, so they do not have in-bound edges. The last two (circle) are blocks, each of which has two inputs. Each input has two choices, Conv or Sep, for example. A mixed operation transforms two choices to continuous space with α_1 and α_2 by a softmax function. Thus, a cell architecture is depicted in the 5(edges)$\times$2($choices$) matrix $\mathbb{A}$. The gradient-based method is applicable to update $\mathbb{A}$ in the continuous space. After being updated, α_1 and α_2 in each edge determine its final operations.

ent networks [74] if they share the same architectures. Child networks could also preserve architectural information from their parents [93] by using network morphism [12]. Both ways avoid training child architecture from scratch, which could save time for evaluation.

5.4 Gradient 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 the 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 [60; 10] as shown in Figure 11. They still make use of the cell-based search space. Given a candidate set of layer operations $\mathcal{O} = \{o_i\}$ and an input x, the output of a mixed operation $m_{\mathcal{O}}(x)$ [60] is a weighted sum of these operations.

$$m_{\mathcal{O}}(x) = \sum_{o \in \mathcal{O}} \frac{\exp(\alpha_o)}{\sum_{o^i \in \mathcal{O}} \exp(\alpha_{o^i})} o(x), \tag{10}$$

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 by a mixed operation, and a DAG cell structure with E edges is then encoded in $E \times |\mathcal{O}|$ architectural weights. The cell architectures are searched by optimizing those architectural weights. Furthermore, the mixed operation can improve its memory usage by binary gates [10], where an operation per edge is sampled according to the probability proportional to its architectural weight. Proximal algorithm [95] plays the same role of binary gates, where we select the operation which has the max architecture 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 (8). 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. Eventually, the largest value of architectural weights for an edge decides its operation.

Another feasible way is using encoder/decoder to learn continuous architecture representation [61]. 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, which maps an embedding to its validation performance in the latent space. The stochastic gradient descent attempts to search the competent embedding representation in the latent space.

5.5 Discussion

AutoDL has been a quick-evolving subfield in AutoML, since Zoph and Le [102] disclosed the feasibility of neural architecture search using reinforcement learning. Less than three years, the burdensome consumption of computing resources has been considerably decreased from thousands of GPU days to less than a half GPU day. The major contribution is the invention of cell-based search space, which becomes the widespread search space in AutoDL. Weight sharing [70] of several sub-graphs also greatly cut the training time from scratch. For the comparison of different AutoDL techniques, EA-based AutoDL, designing flexible base and cell network architectures, has the powerful exploitation among other approaches. But it still consumes more time and resources than other approaches in literature. In addition, thanks to the continues representation of architectural weights [60], we could harness various types of gradient approaches, such as coordinate gradient [85], stochastic variance reduced gradient [43], and Quasi-Newton Methods [21] to optimize architectural weights. Gradient theorems and approaches [8; 4] have developed several decades, so we would enrich the fundamental theory of AutoDL by shifting the gradient paradigm [8] to AutoDL.

6. AUTOML FRAMEWORKS

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

6.1 Automated Feature Engineering

FeatureTools [45] is an open-source framework using Python which can automatically generate features from relational databases. The generation follows relationships described by the schema of a relational database. Given a target entity, the new features come from columns in the forward entity (the target points to) and the backward entity (which points to the target) with arithmetic or aggregate operators. It also provides customers the commercial front-end visualization to examine the synthesized features. Besides, AutoCross [98] 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. Its feature selection ranks the performance of features on a logistic regression regardless of real learning models.

Category	Framework	Language	Provided by	URL
Automated Feature Engineering (AutoFE)	FeatureTools [45]	Python	Open-Source	https://github.com/Featuretools/featuretools
	AutoCross [98]	-	4paradigm	https://www.4paradigm.com
Automated Model and Hyperparameter Tuning (AutoMHT)	Hyperopt [6]	Python	Open-Source	https://github.com/hyperopt/hyperopt
	Scikit-Optimize	Python	Open-Source	https://github.com/scikit-optimize/scikit-optimize
	SMAC3 [39]	Python	Open-Source	https://github.com/automl/SMAC3
	Auto-Sklearn [27]	Python	Open-Source	https://github.com/automl/auto-sklearn
	TPOT [69]	Python	Open-Source	https://github.com/EpistasisLab/tpot
	H2O	Java	H2O.ai	https://github.com/h2oai/h2o-3
	HyperTune	-	Google	https://bit.ly/2IMsECx
	Automatic Model Tuning	-	Amazon	https://aws.amazon.com/sagemaker
	Azure AutoML	-	Microsoft	https://bit.ly/2XxBMmA
	DataRobot AutoML	-	DataRobot	https://bit.ly/331VirD
Automated Deep Learning (AutoDL)	Auto-Keras [42]	Python	Open-Source	https://github.com/keras-team/autokeras
	AdaNet [90]	Python	Open-Source	https://github.com/tensorflow/adanet
	Auto-PyTorch [63]	Python	Open-Source	https://github.com/automl/Auto-PyTorch
	Google AutoML	-	Google	https://cloud.google.com/automl
	AutoGluon	Python	Amazon	https://autogluon.mxnet.io
	Neural Network Intelligence	Python	Microsoft	https://github.com/microsoft/nni

Table 2: A list of selective AutoML Frameworks until March 2020

6.2 Automated Model and Hyperparameter Tuning

Current AutoMHT frameworks are split into hyperparameter tuning and Auto-Pipeline developed from the open-source community or enterprises. For open-source frameworks, Hyperopt [6], SMAC [39], and Scikit-Optimize are tools for the hyperparameter tuning. Hyperopt [6] implements Bayesian optimization (BO) in the form of probability density trees and SMAC [39] implements BO with random forests for the surrogate model. Scikit-Optimize inherits the same interface of GridSearchCV and RandomizedSearchCV in scikit-learn (sklearn). It gives the sklearn community an extra choice to search hyperparameters via the sequential model-based optimization. The surrogate model of BO could be random forests, extra trees and Gaussian process regressions. Additionally, Auto-Sklearn [27], TPOT [69], and H2O are frameworks for Auto-Pipeline. Auto-Sklearn [27] is the extension of SMAC [39] with meta learning and ensemble learning for machine learning pipeline automation. TPOT [69] is the most popular Auto-Pipeline tool using genetic programming to optimize the tree-structured pipeline. H2O is an open-source machine learning platform written in Java developed by H2O.ai. The platform can optimize hyperparameters via random search, grid search, and Bayesian optimization and construct stacking ensembles of pipelines. Its commercial version is H2O Driverless, which could generate versatile features for the pipeline optimization and provide the dashboard to visualize the results.

For enterprise frameworks, Google and Amazon provide cloud services for the hyperparameter tuning, while Microsoft and DataRobot offer Auto-Pipeline frameworks. Google HyperTune [32] could tune hyperparameters of machine learning models by grid and random search, TPE [6], and SMAC [39]. Amazon Automatic Model Tuning could optimize hyperparameters using Bayesian optimization. Furthermore, Azure AutoML uses collaborative filtering and Bayesian optimization to search for promising pipelines [28]. DataRobot integrating H2O libraries provides a commercial Auto-Pipeline platform. Its leaderboard allows customers to identify intermediate model performance and model efficiency during the pipeline optimization.

6.3 Automated Deep Learning

AutoDL frameworks mainly build the architectures of neural networks or tune hyperparameters (e.g., kernel size, number of filters, and learning rate) of given neural networks. Some AutoDL frameworks also consider the two types of automation simultaneously.

Auto-Keras [42] is an open-source project using Keras to search for deep network architectures. It adopts Bayesian optimization and network morphism to search entire arbitrary neural architectures and employs random search and Hyperband [56] to tune non-architectural hyperparameters. Moreover, AdaNet [90] 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. The open-source Auto-Pytorch [63] utilizing BOHB [25] discovers the neural-network-based pipelines from pre-processsors, training optimizer, to network architectures. Its search space includes multilayer perceptron and ResNet [36] with the tunable number of filters in hidden layers.

Besides, Neural Network Intelligence (NNI) is an open-source toolkit from Microsoft for neural architecture search and hyperparameter tuning. It supports several search techniques, e.g., evolutionary algorithm and network morphism, to tune network architectures and provides flexible execution environments in local or cloud servers. Amazon AutoGluon is an open-source framework built on MXNet [13] and implements the controller of ENAS [70] to search the hyperparameters (e.g., kernel size) of MobileNet [38]. Apart from ENAS implementation, AutoGluon also tunes hyperparameters of classification models for tabular datasets by Bayesian optimization or Hyperband [56]. Last but not least, Google Cloud AutoML could get high-quality neural networks on vision, text classification, and language translation. The service is based on the NAS proposed by Zoph and Le [102]. Users could obtain customized models by merely uploading their labeled datasets without any pieces of code.

7. 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. Researchers usually leverage the UCI Machine Learning Repository [2] and OpenML datasets with their setting for evaluation. Unfortunately, no standard protocol for AutoFE provides a fixed set of transformation operations, and identical training setting of generated solutions. In contrast, several benchmarks [23; 3; 30; 101] have been published for AutoMHT for performance comparison.

For AutoDL, NAS-Bench-101 [97] 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 from scratch, which avoids different training techniques and hyperparameters. However, the NAS-Bench-101 [97] only considers the cell-based search space. No benchmark exists for the comparison of AutoDL within the whole-network search space. Authoritative benchmarks for AutoML are required; otherwise, it would be hard to compare the effectiveness and efficiency among a multitude of AutoML frameworks.

Efficiency is one of the dominant factors for the successful adoption of AutoML methods. Some AutoML methods take tolerable time to generate solutions: Auto-Sklearn [27] requires 500 seconds in a CPU to find competent pipelines; FeatureRL [49] takes 400 seconds in a CPU to find appropriate features for a learning model. In contrast, AutoDL usually demands massive computation resources and time, e.g., thousands of GPU days [102]. Novel acceleration techniques are proposed to reduce the enormous order of time [42; 70; 60; 96]. Specifically, NASP [96] only took less than half GPU-day to locate architectures that attain competitive performance on CIFAR-10 [53]. Aside from the search efficiency of AutoDL, the evaluation stage that retrains the searched neural networks with large layers and epochs also takes massive time. For instance, we ran NASP in our machine, where we spent nine times more evaluating the optimal architecture (1.9 GPU days) than discovering it (0.2 GPU days), The two efficiency issues are still open problems in AutoDL.

The design of search spaces is another challenge for AutoML, which requires substantial knowledge and experience from human experts, and is the portion far from automation. AutoML liberates the community from manual tuning models, but understanding the underlying theory and characteristics of machine learning is inevitable. A practical search space incorporates experienced human prior knowledge for machine learning and considers the trade-off between its size (compact) and all useful possibilities (comprehensive). We believe that appropriate search space is more crucial than optimization approaches. A well-known example is the cell-based search [103] for AutoDL. The repeated neural network blocks in ResNet [36] or DenseNet [40] motivate researchers to fix the base network architecture and to search cell structures only. The cell-based search space has succeeded in RL [70], EA [59], and gradient descent [60]. The handcrafted search spaces highly affect the performance of AutoML, which is still an intriguing question.

Interpretable analysis can provide insight on what crucial search components will lead to desired performance. For instance, it can tell human experts learn why features are chosen, and why architectures evolve in such manner. With the integration of friendly user interfaces [57], AutoML users could also customize their solutions by selecting desirable configurations between independent runs. Thus, it would spark experts to devise suitable search space, and make innovative neural network design easier. Nevertheless, any AutoML search technique is still a black-box procedure. Insufficient search knowledge from AutoML could be utilized. To advance the AutoML interpretability, we could learn from techniques of interpretable machine learning [22]. We believe interpretability and user interaction can increase the usefulness and capability of future AutoML.

Miscellaneous domains remain potential to collaborate with AutoML. Most AutoML frameworks have focused on classical regression and classification problems. Particularly, AutoFE and AutoMHT are developed for UCI [2] and Kaggle datasets. Empirical results of AutoDL are majorly from image classification on CIFAR [53] and ImageNet [20] or from language modeling on Penn Treebank [62] and WikiText-2 [64]. Recently, AutoDL has been developed to deal with segmentation [58; 91], graph-CNN [29; 100], GAN [33], and autoencoders [80]. One could make AutoDL robust to adversarial attack [18] and label noise [15]. Moreover, AutoML has gotten along with recommendation system [95], in which one discovered the neural interaction functions for collaborative filtering. Other distinct machine learning applications, like anomaly detection, real-time semantic segmentation, action detection in video, are also worthy of working together with AutoML. We believe the cooperation of AutoML with multiple potential domains could become another promising research direction.

8. 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. In this work, we reveal AutoML is a bi-level optimization problem and generalize AutoML heuristic techniques from the iterative solver, even though AutoML is an NP-complete problem. When researchers want to develop their AutoML algorithms/tools/services, they need to take account of the three questions, (1) what category they want to deal with, e.g., AutoFE, AutoMHT, or AutoDL; (2) what search scope of the category is, e.g., the range of hyperparameters; and (3) what technique is appropriate to perform the search. By including and introducing both mainstream AutoML techniques and selected AutoML frameworks, we hope this survey could help researchers to answer the above questions and could give insight into the progress of AutoML.

9. ACKNOWLEDGEMENTS

This work is, in part, supported by DARPA under grant #FA8750-17-2-0116 and by NSF under grant #IIS-1750074. The views, opinions, and/or findings expressed are those of the authors and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government. We would like to thank Qingquan Song for proofreading the paper and thank Xia Hu for the helpful discussions of the paper structure. We also thank Hanghang Tong and other reviewers for their useful comments.

10. REFERENCES

[1] Amazon. Perform automatic model tuning. `https://docs.aws.amazon.com/en_us/sagemaker/latest/dg/automatic-model-tuning.html`. Accessed: 2020-02-21.

[2] A. Asuncion and D. Newman. Uci machine learning repository, 2007.

[3] A. Balaji and A. Allen. Benchmarking automatic machine learning frameworks. *arXiv preprint arXiv:1808.06492*, 2018.

[4] T. Ben-Nun and T. Hoefler. Demystifying parallel and distributed deep learning: An in-depth concurrency analysis. *ACM Computing Surveys (CSUR)*, 52(4):1–43, 2019.

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

[6] 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.

[7] A. Blum and R. L. Rivest. Training a 3-node neural network is np-complete. In *Advances in neural information processing systems*, pages 494–501, 1989.

[8] L. Bottou, F. E. Curtis, and J. Nocedal. Optimization methods for large-scale machine learning. *Siam Review*, 60(2):223–311, 2018.

[9] J. Bourgain. On lipschitz embedding of finite metric spaces in hilbert space. *Israel Journal of Mathematics*, 52(1-2):46–52, 1985.

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

[11] 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.

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

[13] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*, 2015.

[14] X. Chen, Q. Lin, C. Luo, X. Li, H. Zhang, Y. Xu, Y. Dang, K. Sui, X. Zhang, B. Qiao, et al. Neural feature search: A neural architecture for automated feature engineering. In *2019 IEEE International Conference on Data Mining (ICDM)*, pages 71–80. IEEE, 2019.

[15] Y.-W. CHEN, Q. Song, X. Liu, P. Sastry, and X. Hu. On robustness of neural architecture search under label noise. *Frontiers in Big Data*, 3:2, 2020.

[16] B. Colson, P. Marcotte, and G. Savard. An overview of bilevel optimization. *Annals of operations research*, 153(1):235–256, 2007.

[17] P. Covington, J. Adams, and E. Sargin. Deep neural networks for youtube recommendations. In *Proceedings of the 10th ACM conference on recommender systems*, pages 191–198, 2016.

[18] E. D. Cubuk, B. Zoph, S. S. Schoenholz, and Q. V. Le. Intriguing properties of adversarial examples. *arXiv preprint arXiv:1711.02846*, 2017.

[19] U. Demšar, P. Harris, C. Brunsdon, A. S. Fotheringham, and S. McLoone. Principal component analysis on spatial data: an overview. *Annals of the Association of American Geographers*, 103(1):106–128, 2013.

[20] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR09*, 2009.

[21] J. E. Dennis, Jr and J. J. Moré. Quasi-newton methods, motivation and theory. *SIAM review*, 19(1):46–89, 1977.

[22] M. Du, N. Liu, and X. Hu. Techniques for interpretable machine learning. *Communications of the ACM*, 63(1):68–77, 2019.

[23] 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.

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

[25] S. Falkner, A. Klein, and F. Hutter. Bohb: Robust and efficient hyperparameter optimization at scale. *arXiv preprint arXiv:1807.01774*, 2018.

[26] M. Feurer and F. Hutter. *Hyperparameter Optimization*, pages 3–33. Springer International Publishing, Cham, 2019.

[27] 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.

[28] N. Fusi, R. Sheth, and M. Elibol. Probabilistic matrix factorization for automated machine learning. In *Advances in Neural Information Processing Systems*, pages 3348–3357, 2018.

[29] Y. Gao, H. Yang, P. Zhang, C. Zhou, and Y. Hu. Graphnas: Graph neural architecture search with reinforcement learning. *arXiv preprint arXiv:1904.09981*, 2019.

[30] P. Gijsbers, E. LeDell, J. Thomas, S. Poirier, B. Bischl, and J. Vanschoren. An open source automl benchmark. *arXiv preprint arXiv:1907.00909*, 2019.

[31] D. E. Goldberg and K. Deb. A comparative analysis of selection schemes used in genetic algorithms. In *Foundations of genetic algorithms*, volume 1, pages 69–93. Elsevier, 1991.

[32] D. Golovin, B. Solnik, S. Moitra, G. Kochanski, J. Karro, and D. Sculley. Google vizier: A service for black-box optimization. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '17, pages 1487–1495, New York, NY, USA, 2017. ACM.

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

[34] Google. Overview of hyperparameter tuning. `https://cloud.google.com/ai-platform/training/docs/hyperparameter-tuning-overview`. Accessed: 2020-02-21.

[35] A. Graves, A.-r. Mohamed, and G. Hinton. Speech recognition with deep recurrent neural networks. In *2013 IEEE international conference on acoustics, speech and signal processing*, pages 6645–6649. IEEE, 2013.

[36] 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*, pages 770–778, 2016.

[37] X. He, K. Zhao, and X. Chu. Automl: A survey of the state-of-the-art. *arXiv preprint arXiv:1908.00709*, 2019.

[38] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.

[39] 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.

[40] F. Iandola, M. Moskewicz, S. Karayev, R. Girshick, T. Darrell, and K. Keutzer. Densenet: Implementing efficient convnet descriptor pyramids. *arXiv preprint arXiv:1404.1869*, 2014.

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

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

[43] R. Johnson and T. Zhang. Accelerating stochastic gradient descent using predictive variance reduction. In *Advances in neural information processing systems*, pages 315–323, 2013.

[44] 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.

[45] 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.

[46] 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.

[47] 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.

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

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

[50] 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.

[51] 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.

[52] B. Korte, J. Vygen, B. Korte, and J. Vygen. *Combinatorial optimization*, volume 2. Springer, 2012.

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

[54] H. T. Lam, J.-M. Thiebaut, M. Sinn, B. Chen, T. Mai, and O. Alkan. One button machine for automating feature engineering in relational databases. *arXiv preprint arXiv:1706.00327*, 2017.

[55] H. O. Lancaster and E. Seneta. Chi-square distribution. *Encyclopedia of biostatistics*, 2, 2005.

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

[57] R. Liaw, E. Liang, R. Nishihara, P. Moritz, J. E. Gonzalez, and I. Stoica. Tune: A research platform for distributed model selection and training. *arXiv preprint arXiv:1807.05118*, 2018.

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

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

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

[61] 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.

[62] M. P. Marcus, B. Santorini, M. A. Marcinkiewicz, and A. Taylor. Treebank-3. *Linguistic Data Consortium, Philadelphia*, 14, 1999.

[63] H. Mendoza, A. Klein, M. Feurer, J. T. Springenberg, M. Urban, M. Burkart, M. Dippel, M. Lindauer, and F. Hutter. Towards automatically-tuned deep neural networks. In F. Hutter, L. Kotthoff, and J. Vanschoren, editors, *AutoML: Methods, Sytems, Challenges*, chapter 7, pages 141–156. Springer, Dec. 2018. To appear.

[64] S. Merity, C. Xiong, J. Bradbury, and R. Socher. Pointer sentinel mixture models. *arXiv preprint arXiv:1609.07843*, 2016.

[65] Microsoft. What is automated machine learning? `https://docs.microsoft.com/en-us/azure/machine-learning/concept-automated-ml`. Accessed: 2020-02-21.

[66] D. Mladenic. Machine learning on non-homogeneous, distributed text data. *Computer Science, University of Ljubljana, Slovenia*, 1998.

[67] F. Nargesian, H. Samulowitz, U. Khurana, E. B. Khalil, and D. S. Turaga. Learning feature engineering for classification. In *IJCAI*, pages 2529–2535, 2017.

[68] 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.

[69] 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.

[70] 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, Stockholmsmässan, Stockholm Sweden, 10–15 Jul 2018. PMLR.

[71] 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.

[72] 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.

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

[74] E. Real, S. Moore, A. Selle, S. Saxena, Y. L. Suematsu, J. Tan, Q. V. Le, and A. Kurakin. Large-scale evolution of image classifiers. In D. Precup and Y. W. Teh, editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 2902–2911, International Convention Centre, Sydney, Australia, 06–11 Aug 2017. PMLR.

[75] F. Sebastiani. Machine learning in automated text categorization. *ACM computing surveys (CSUR)*, 34(1):1–47, 2002.

[76] B. Shahriari, K. Swersky, Z. Wang, R. P. Adams, and N. De Freitas. Taking the human out of the loop: A review of bayesian optimization. *Proceedings of the IEEE*, 2016.

[77] A. Sharma and K. K. Paliwal. Linear discriminant analysis for the small sample size problem: an overview. *International Journal of Machine Learning and Cybernetics*, 6(3):443–454, 2015.

[78] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, et al. Mastering the game of go without human knowledge. *Nature*, 550(7676):354–359, 2017.

[79] J. Snoek, H. Larochelle, and R. P. Adams. Practical bayesian optimization of machine learning algorithms. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 2951–2959. Curran Associates, Inc., 2012.

[80] M. Suganuma, M. Ozay, and T. Okatani. Exploiting the potential of standard convolutional autoencoders for image restoration by evolutionary search. *arXiv preprint arXiv:1803.00370*, 2018.

[81] R. S. Sutton and A. G. Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[82] R. S. Sutton, D. A. McAllester, S. P. Singh, and Y. Mansour. Policy gradient methods for reinforcement learning with function approximation. In *Advances in neural information processing systems*, pages 1057–1063, 2000.

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

[84] 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.

[85] P. Tseng and S. Yun. A coordinate gradient descent method for nonsmooth separable minimization. *Mathematical Programming*, 117(1-2):387–423, 2009.

[86] V. N. Vapnik. *Statistical Learning Theory*. Wiley-Interscience, New York, NY, 1998.

[87] J. R. Vergara and P. A. Estévez. A review of feature selection methods based on mutual information. *Neural computing and applications*, 24(1):175–186, 2014.

[88] F. Viegas, L. Rocha, M. Gonçalves, F. Mourão, G. Sá, T. Salles, G. Andrade, and I. Sandin. A genetic programming approach for feature selection in highly dimensional skewed data. *Neurocomputing*, 273:554–569, 2018.

[89] C. Villani. *Optimal transport: old and new*, volume 338. Springer Science & Business Media, 2008.

[90] 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.

[91] Y. Weng, T. Zhou, Y. Li, and X. Qiu. Nas-unet: Neural architecture search for medical image segmentation. *IEEE Access*, 7:44247–44257, 2019.

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

[93] M. Wistuba. Deep learning architecture search by neuro-cell-based evolution with function-preserving mutations. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 243–258. Springer, 2018.

[94] M. Wistuba, A. Rawat, and T. Pedapati. A survey on neural architecture search. *arXiv preprint arXiv:1905.01392*, 2019.

[95] Q. Yao, X. Chen, J. T. Kwok, Y. Li, and C.-J. Hsieh. Efficient neural interaction function search for collaborative filtering.

[96] Q. Yao, J. Xu, W.-W. Tu, and Z. Zhu. Differentiable neural architecture search via proximal iterations. *arXiv preprint arXiv:1905.13577*, 2019.

[97] 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.

[98] 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.

[99] J. Zhang, Z.-h. Zhan, Y. Lin, N. Chen, Y.-j. Gong, J.-h. Zhong, H. S. Chung, Y. Li, and Y.-h. Shi. Evolutionary computation meets machine learning: A survey. *IEEE Computational Intelligence Magazine*, 6(4):68–75, 2011.

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

[101] M.-A. Zöller and M. F. Huber. Benchmark and survey of automated machine learning frameworks.

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

[103] B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le. Learning transferable architectures for scalable image recognition. June 2018.