MOAZ: A Multi-Objective AutoML-Zero Framework

Ritam Guha[1, 3], Wei Ao[1], Stephen Kelly[2], Vishnu Boddeti[1], Erik Goodman[1], Wolfgang Banzhaf[1], Kalyanmoy Deb[1, 3]
Michigan State University, East Lansing, Michigan, USA[1]
McMaster University, Hamilton, Ontario, Canada[2]
Computational Optimization and Innovation (COIN) Laboratory, USA[3]
{guharita,aowei}@msu.edu,kellys32@mcmaster.ca,{vishnu,goodman,banzhafw,kdeb}@msu.edu

COIN Report Number 2023005

ABSTRACT

Automated machine learning (AutoML) greatly eases human efforts in architecture engineering. However, mainstream AutoML methods like neural architecture search (NAS) are customized for well-designed search spaces wherein promising architectures are densely distributed. In contrast, AutoML-Zero builds machine-learning algorithms using basic primitives and can explore novel architectures beyond human knowledge. AutoML-Zero shows the potential to deploy machine learning systems by not taking advantage of either feature engineering or architectural engineering. In its current form, it only optimizes a single objective like accuracy and has no mechanism to ensure that the constraints of real-world applications are satisfied. We propose a multi-objective variant of AutoML-Zero called MOAZ, that distributes solutions on a Pareto front by trading off accuracy against the computational complexity of the machine learning algorithm. In addition to generating different Pareto-optimal solutions, MOAZ can effectively explore the sparse search space to improve search efficiency. Experimental results on linear regression tasks show MOAZ reduces the median complexity by 87.4% compared to AutoML-Zero while accelerating the median target performance achievement speed by 82%. In addition, our preliminary results on non-linear regression tasks show the potential for further improvements in search accuracy and for reducing the need for human intervention in AutoML.

KEYWORDS

AutoML, Evolutionary Algorithms, Multi-objective search.

1 INTRODUCTION

Two broad considerations guide design of Machine Learning (ML) algorithms—building predictive models that: 1) optimize an objective function, while 2) minimizing computational complexity. While a model's predictive power is the primary measure of quality, the energy required to build and deploy the model is a common engineering constraint for real-world systems and, from a broader societal perspective, is a critical consideration given the current climate crisis [32]. Efficiency is especially critical in the development of deployable algorithms because 1) each model is executed multiple times at scale, and 2) for models deployed on low-power embedded hardware [15], it is critical to consider the computational complexity of the model. So, it is important to treat ML development as a multi-objective problem, where one objective is the predictive accuracy of a model and the other objective is its computational requirement. Additional objectives can also be introduced depending on the application area.

In recent years, ML has benefited greatly from a transition from feature engineering [10] to architecture engineering [11]. For example, ResNets [22], Transformers [14], and Graphical Neural Networks [24] are designed for processing vision, language, and graphical data, but designing the associated architecture is a trial-and-error procedure which requires much human effort. Automated machine learning (AutoML) [13, 16, 17] is proposed to reduce this need for human effort. The goals of AutoML are to make ML more accessible to non-ML experts, improve the efficiency of ML systems, and accelerate research in Artificial Intelligence (AI) application development [18]. Automating a search through the vast space of algorithms is a daunting task well suited to Evolutionary Algorithms (EAs), in particular, Genetic Programming (GP) [3, 20], and a variety of powerful methods have recently appeared in the literature [7, 21, 23, 26, 28]. AutoML-Zero (AZ) [28] has shown how, starting with only basic mathematical operations as building blocks, EAs can build complete machine learning algorithms that incorporate widely-used, expert-designed techniques such as backpropagation. The method has initially been demonstrated under simple regression problems and well-known image classification benchmarks such as CIFAR-10. The objectives of this study are to expand the breadth of real-life applicability for AZ by showing how the method can be applied to ML problems and explicitly address computation simplicity and efficiency in the search process. The effect of multi-objective AutoML has also been studied by Pfisterer et. al in [25] which proposed a multi-objective income prediction pipeline with the two objectives of misclassification error and fairness.

We have replaced the search algorithm of AZ with a modified version of NSGA-II [9]. The basic version of NSGA-II was updated to support the needs of algorithm development in the AZ framework. We have named the new framework Multi-Objective AZ (MOAZ). In this paper, we demonstrate that making the search algorithm multi-objective yields two improvements over AZ: 1) the complexity of the resulting algorithms becomes much smaller compared to algorithms discovered by AZ, and 2) the success rate of the algorithm search improves significantly. In each iteration of a multi-objective search, it maintains a diversity of the discovered solutions in terms of different objectives, which in turn improves the search efficiency and helps to discover better solutions than single-objective variants with a similar amount of effort.

The rest of the paper is organized as follows: Section 2 gives a brief introduction to the AZ framework. The main modifications

over AZ used to build MOAZ are provided in Section 3. The experimental analysis of the application of MOAZ on a linear regression problem and some preliminary results on a non-linear regression problem is presented in Section 4. Finally, the paper is concluded in Section 5.

2 AUTOML-ZERO (AZ) APPROACH

In this section, we have described the AZ approach, which is a precursor to this work, in a generic AutoML setting. The AZ process introduced by Real et. al in [28] exhibits the power of EAs to discover from scratch fully functional ML algorithms by using simple operations. Any AutoML process can be defined by its solution representation, search space, search process, and evaluation strategy. We can define AZ in this format as follows:

Solution Representation: Any solution in AZ is an ML algorithm represented by virtual register machine instructions like in Linear Genetic Programming [4], but manipulating more complex data structures (scalars, vectors, or matrices). Following the basic structure of supervised ML algorithms, it divides an algorithm into three components: `Setup`, `Predict`, and `Learn`. The job of the search algorithm is to populate these components with appropriate instructions.

Search Space: The AZ search space consists of 65 fundamental operations like addition, subtraction, multiplication, etc. These instructions are used in sequence to construct each of the components of an AZ solution. A list of all the operations is mentioned in the supplementary material. The search algorithm must be efficient enough to combine those simple operations to synthesize complicated ML algorithms.

Search Algorithm: The authors of [28] have used Regularized Evolution (RE) [27], which is a popular EA for single-objective optimization tasks.

Evaluation: As AZ attempts to synthesize ML algorithms, the resulting algorithms should have the ability to learn from existing data and generalize to new data. To evaluate the generalization ability of the searched algorithms over regression, AZ creates multiple train and test datasets with random data samples. Each solution algorithm is trained and tested on each of these datasets and its mean test accuracy serves as the fitness for the solution algorithm.

3 MULTI-OBJECTIVE AUTOML-ZERO (MOAZ) APPROACH

There are several improvements to the existing AZ approach that can be achieved through a multi-objective formulation, which we discuss in the following:

- The AZ framework uses a **single-objective evolutionary algorithm** called Regularized Evolution (RE) [27]. However, most real-world problems are multi-objective ($2-3$ objectives) [8] or many-objective (>3 objectives) [12] in nature. As RE is dealing with a single objective, there is no explicit operation to maintain diversity in the population. So, it can easily get **stuck in local optima** and the only way to avoid it is to restart the process. AZ typically uses several restarts to escape from local optima.
- The ultimate goal of developing such algorithmic search procedures is to reduce the need for human intervention. It is a well-known fact that the stochasticity of GP leads to redundancy in final solutions [1, 2, 20, 31]. In linear GP (LGP) systems, algorithms exist to extract the relevant semantics of a genetic program [4]. This can be done for each individual prior to execution and saves computational effort. However, in earlier work on AZ this was not done, either because it is more complicated due to the structure or considered not worthwhile. As a result, the final algorithms had **many redundant instructions** which had to be manually analyzed and removed from the algorithms before determining their functions. This final step requires significant human intervention. While there are also some post-processing tools for LGP [6, 30] that can analyze the syntax and semantics of resulting algorithms and remove ineffective instructions after a final algorithm has been discovered, here we pursue a different avenue and develop a multi-objective approach.

The goal of the method proposed here is to modify the basic AZ framework to make it applicable to more realistic situations involving multiple conflicting objectives and also to reduce the computational load and redundancy of the discovered algorithms. For this purpose, we have replaced the search algorithm in AZ with a modified version of NSGA-II.

3.1 Formulating Multiple Objectives

When we are dealing with a single-objective optimization problem, the optimizer considers only one objective. So, two solutions having the same objective scores are treated in exactly the same way. For example, if a simple algorithm and a complicated algorithm are providing the same accuracy, they cannot be differentiated and the search process might arbitrarily choose the complicated algorithm. For this reason, in the proposed approach, we are adding a new objective to the search process which is a measure of the computational complexity of the algorithms. Thus, the problem turns into a bi-objective optimization problem. Adding an additional objective has more than one benefit. Past work [5, 19] have shown that converting a single-objective optimization problem into a multi-objective version can be more efficient in avoiding local optima and improving the speed of reaching optimal solutions.

3.2 Objective Definition

The two objectives used for the search process are *accuracy* and *computational complexity*, where we want to maximize the accuracies and minimize the computational complexities of the ML algorithms. As a standard practice in multi-objective optimization, we convert both objectives to minimization objectives by replacing the accuracy with the error rate. The final optimization formulation then becomes:

$$\begin{aligned} \min_{x \in S} \quad & \{error(x),\ complexity(x)\}, \\ \text{subject to} \quad & \text{lb}_{error} \leq error(x) \leq \text{ub}_{error}, \\ & \text{lb}_{complexity} \leq complexity(x) \leq \text{ub}_{complexity}. \end{aligned} \tag{1}$$

where lb_{error} and ub_{error} represent the lower and upper bounds on the acceptable error of the ML algorithms, while $\text{lb}_{\text{complexity}}$ and $\text{ub}_{\text{complexity}}$ represent lower and upper bounds on the acceptable complexity. These constraints on the objectives become important for multi-objective optimization because we are not really interested

in solutions having very high errors or complexity. So we add these constraints to generate more solutions in the region of interest. It is not necessary to add lower bounds for the objectives that we have used, but we wanted to make the formulation more generalistic as other objectives might benefit from having a lower bound. In order to compute the complexity of the ML algorithms, we first approximate the FLOPs required by every basic operation. Then the model complexity can be defined as the sum of the FLOPs used by the operations of an algorithm.

3.3 Search Algorithm

MOAZ uses the same search space and solution representation as AZ, but when we are working with multiple objectives, we need more sophisticated algorithms to maintain a diversified set of solutions to find trade-off points along all the objectives. NSGA-II is one of the most popular multi-objective optimization algorithms in the literature. It is a modular algorithm with different components which can interact independently of each other. This property makes it highly customizable and extendable to different problems. In this work, we have customized NSGA-II's selection operation—more specifically, the dominance checking operation, and the crossover operator, to make it applicable in MOAZ. Typically, mutation serves as a local search process and in most cases, the child solutions are close to the parent solutions after mutation. In order to maintain sufficient diversity in the population, we need to add a crossover operation along with mutation. In the following subsections, we have explained the modifications suggested for NSGA-II to make it applicable to the MOAZ framework.

3.3.1 Domination Checking. Domination checking is one of the most important operations in NSGA-II. It is used extensively across the algorithm for comparing candidate solutions and dividing them into different quality fronts. In the context of MOAZ, we need to compare two ML algorithms with respect to two criteria: *error* and *complexity*. One of the interesting intuitions we used to modify the search is that out of the three components of AZ algorithms, the `Predict` component is called most often, followed by `Learn` and finally `Setup`. In deployment, when the setup and learning are over, the trained algorithm is just used for inference over multiple inputs and the `Predict` component is the only component that gets used after deployment. So, we can assign importance to reduce the complexity of these components in the following order: `Predict` $\rightarrow$ `Learn` $\rightarrow$ `Setup`. Intuitively, a working algorithm with simpler `Predict` will be desired over another working algorithm with more complex `Predict` because `Predict` is used much more often than the other components. That is why we have added a lexicographic comparison for the complexities of the different components. The constraint violations for each algorithm are computed using Algorithm 1, where the $\langle x \rangle$ operator indicates $\max(x, 0)$. Algorithm 2 illustrates the strategy to perform lexicographical complexity comparison between a pair of algorithms. Finally, the overall constrained domination check is performed according to Algorithm 3. In Lines 4-15, we compare two algorithms when at least one of them is violating constraints mentioned in Equation (1). If one of the algorithms is violating constraints and the other one is not, the latter algorithm is said to dominate the former one. If both the algorithms are violating constraints, the algorithm with the lower constraint violation score obtained from Algorithm 1 is said to dominate the other one. Lines 16-32 represent the situation when both algorithms are not violating any constraint. Then if one algorithm has better objective values compared to the other one, the first algorithm is said to dominate the other one, else both of them become non-dominating to each other.

3.3.2 Component Crossover. As different instructions in an algorithm are linked together, it is important to maintain the linkage and come up with a recombination operation (crossover+mutation) that can maintain the associativity among the instructions (*instruction-level linking*) as well as across different components (*component-level linking*). If we use a crossover at the instruction level (e.g. crossing instruction 5 of parent 1 with instruction 5 of parent 2), it becomes difficult to maintain both linkages because we do not know if the instructions being recombined are doing similar functions in the two algorithms. Moreover, the sizes of the algorithms can be different and this makes it difficult to perform instruction-level crossovers. One approach to maintaining the linkage among instructions and ensuring that the sizes of the models do not become an issue is to perform crossover at the component level. The idea for the component-level crossover is to cross components from different parents (e.g. `Setup` from parent 1 with the `Setup` from parent 2). This ensures that instruction-level linkage in a single component function remains unaltered. On the other hand, as the component functions are having similar goals for both algorithms, the component-level linkage is better compared to a purely positional approach. We expect the mutation operator to take care of the *instruction-level linkage.* MOAZ uses the same mutation strategies as AZ (e.g. insert instructions, remove instructions, alter parameters, etc.). So for example, if a particular instruction is not useful in an algorithm, it can be removed using a mutation operation.

Algorithm 1 Constraint Violation Computation (CVC) . $\langle \alpha \rangle = \alpha$, if $\alpha > 0$; zero, otherwise.

Input: fit, lb_{error}, ub_{error}, lb_{comp}, ub_{comp}
Output: cv
Initialization: $cv \leftarrow 0$
1: $cv \leftarrow 0$
2: $cv \leftarrow cv + \langle fit.error - \text{ub}_{error} \rangle + \langle \text{lb}_{error} - fit.error \rangle$
3: $cv \leftarrow cv + \langle fit.comp - \text{ub}_{comp} \rangle + \langle \text{lb}_{comp} - fit.comp \rangle$
4: **return** cv

4 EXPERIMENTS

In this section, we have explained some of the empirical results we obtained while applying the MOAZ framework to linear regression problems. The results obtained are mainly compared with those of the basic AZ framework. Finally, we also show some results on a non-linear problem [1].

4.1 Linear Regression Problem Setup

For testing MOAZ, we have formulated an d-dimensional linear regression problem. The dataset for the linear regression problem

[1]Implementation details for these experiments can be found in the following Github repository: https://github.com/Ritam-Guha/moaz.

Algorithm 2 Lexicographic Comparison of Complexity (*LCC*)

Input: C_1, C_2 {C_i denotes complexity of algorithm i}
Output: $flag$

```
if C1.Predict < C2.Predict then
    flag ← 1
else if C1.Predict > C2.Predict then
    flag ← −1
else
    if C1.Learn < C2.Learn then
        flag ← 1
    else if C1.Learn > C2.Learn then
        flag ← −1
    else
        if C1.Setup < C2.Setup then
            flag ← 1
        else if C1.Setup > C2.Setup then
            flag ← −1
        else
            flag ← 0
        end if
    end if
end if
return flag
```

Algorithm 3 Customized dominance check

Input: $fit_1, fit_2, \text{lb}_{error}, \text{ub}_{error}, \text{lb}_{complexity}, \text{ub}_{complexity}$
Output: $dominance$

```
# run Algorithm 1 over the fitnesses to compute the constraint violations.
cv1 ← CVC(fit1, lb_error, ub_error, lb_comp, ub_comp)
cv2 ← CVC(fit2, lb_error, ub_error, lb_comp, ub_comp)
# run Algorithm 2 over the fitnesses to get the lexicographic comparison results.
lex_comp ← LCC(fit1.comp, fit2.comp)
# compare the algorithms for deciding dominance.
if cv1 ≤ 0 and cv2 > 0 then
    dominance ← 1
else if cv1 > 0 and cv2 ≤ 0 then
    dominance ← −1
else if cv1 > 0 and cv2 > 0 then
    if cv1 = cv2 then
        dominance ← 0
    else if cv1 < cv2 then
        dominance ← 1
    else
        dominance ← −1
    end if
else
    if fit1.error < fit2.error then
        if lex_comp = 1 || lex_comp = 0 then
            dominance ← 1
        else
            dominance ← 0
        end if
    else if fit1.error = fit2.error then
        dominance ← lex_comp
    else
        if lex_comp = −1 || lex_comp = 0 then
            dominance ← −1
        else
            dominance ← 0
        end if
    end if
end if
return dominance
```

is created by sampling some random d-dimensional data points and corresponding d-dimensional weight vectors from the standard Gaussian distribution $\mathcal{N}(0, 1)$. The weights are then multiplied by the corresponding data points to create the actual labels of the regression problem. The number of training examples and validation examples for each dataset is pre-specified as 1,000 and 100, respectively. The process is run on 10 such datasets to further analyze the generalization capabilities of the found algorithms. The final performance of an algorithm is defined as the mean of the test errors computed over the 10 datasets. For our experiments, we initially used the linear regression dimension as d=4.

For both MOAZ and AZ, the process stops when an algorithm is discovered in a run that provides a mean target error (e_T) of 10^{-1}. If no such algorithm is found in a run, the process is restarted with a different initial random population. At most 9 such restarts are allowed for the process, making the maximum number of runs 10. The success rate ($\mathbb{S}_r$) of a process is defined as the percentage of times the process is able to find an acceptable algorithm. The number of evaluated algorithms to success ($\mathbb{S}_c$) is the number of algorithms evaluated before discovering an acceptable algorithm for the first time.

4.2 Application to Linear Regression

4.2.1 AZ Solution. If we use one run for the basic AZ framework for solving the linear regression problem, we obtain the solution shown in Figure 1 that has a complexity of 136 as computed by the proposed complexity measurement process. Although it is a perfectly working solution, the question to ask is whether the algorithm needs that many instructions to solve a simple linear regression problem. The main objective function driving the search in AZ is the error of the algorithm. If two algorithms having different complexities are evaluated, they will have the same objective value. So, there is no added incentive to bias the search toward low-complexity solutions. That is why the final solution of the process ends up looking unnecessarily complicated. Moreover, upon careful observation, it is clear that this solution includes many redundant operations and is much harder to analyze due to this complexity.

4.2.2 MOAZ Solution. For applying MOAZ to linear regression, we must specify upper and lower bounds for complexity and error values. We have used an upper bound of 0.6 for error and 100 for complexity. The lower bounds are set as 0 for both. At the end of each run, we expect to get a set of algorithms having errors in the range [0, 0.6] and complexities in [0, 100].

After obtaining the Pareto front for the run, we select the solution with the least complexity with sufficient error as the preferred solution for the problem. One of these solutions can be represented in an algorithmic format as shown in Figure 2. The complexity of the algorithm is 28, which is much lower than the solution obtained by AZ. Even for this solution, there are some redundant operations that can be removed after manual analysis. But as we are now dealing with an algorithm of complexity 28, it becomes easier to analyze than in the AZ scenario. We can clearly see that the algorithm has incorporated a backpropagation-like structure in the `Learn` component. It uses a learning rate of 0.32291 which is stored in $s3$. The algorithm computes the error between the prediction ($s1$) and label ($s0$) and stores it in $s2$ memory. The corresponding gradients get stored in $v2$ which then gets added to the weight vector $v1$ in the weight update stage. As the complexity of the resulting algorithm is low, it was easier to analyze.

```
# sX/vX: scalar/vector memory
# at address X.
def Setup():
 v2 = s2 * v0
 s3 = s0 * s0
 s2 = dot(v1, v2)
 s2 = dot(v1, v1)
 s1 = 0.0323415
 s2 = -0.885379
 s2 = s0 * s2
 s3 = s2 * s0
 s2 = dot(v2, v1)
def Predict(v0):
 v2 = v2 + v1
 s3 = dot(v2, v2)
 v1 = v0 + v2
 v1 = s3 * v0
 s1 = dot(v2, v0)
 s1 = dot(v0, v2)
 s2 = dot(v1, v2)
 s2 = dot(v0, v1)
 v1 = s0 * v1
 s3 = dot(v1, v0)
def Learn(v0, s0):
 s3 = dot(v2, v1)
 s3 = s2 - s1
 v1 = v1 + v2
 s3 = s1 - s0
 s1 = dot(v1, v1)
 v2 = v2 + v1
 v2 = v2 + v2
 s2 = -0.127324
 v2 = s3 * v0
 v2 = s2 * v2
```

Figure 1: Illustration of one of the linear regression algorithms found by the basic AZ framework with a complexity of 136. Both the training and testing error for this algorithm is 0.

```
# sX/vX: scalar/vector memory
# at address X.
def Setup():
 s2 = s0 - s3
 s1 = s2 * s1
 s1 = s1 * s3
 s3 = s2 * s1
 s1 = s0 * s2
def Predict(v0):
 # v0: data sample
 s1 = s0 - s0
 s2 = s2 * s3
 s1 = dot(v0, v1) # prediction
 s2 = s3 - s1
 s3 = 0.32291
def Learn(v0, s0):
 # v0: data sample
 # s0: label
 s2 = s0 - s1 # compute error
 s3 = s2 * s3
 v2 = s3 * v0 # gradient
 v1 = v2 + v1 # weight update
 s3 = s0 - s2
```

Figure 2: Illustration of one of the linear regression algorithms found by the MOAZ framework with a complexity of 28. This algorithm also has 0 training and test errors.

4.3 Analysis on Multiple Runs

In the case of EAs, it is standard practice to apply the process multiple times to the same problem and to draw conclusions from multiple runs, due to the stochasticity of the process. In the present situation, both AZ and MOAZ have been applied to the linear regression problems 30 times. We have combined all the algorithms obtained for AZ and MOAZ in Figure 11 and Figure 12, respectively. The final non-dominated points are colored differently. From the plots, it can be seen that both frameworks can provide workable solutions in the end, but MOAZ solution complexities are lower compared to those of AZ. The solutions for both AZ and MOAZ have been compared in Table 1. There we can see that AZ is able to converge (discover an algorithm with $\leq 10^{-1}$ error) approximately 67% ($\mathbb{S}_r = 66.67$) of the time, whereas MOAZ is able to get success 100% of the time ($\mathbb{S}_r = 100$). Moreover, the median complexities of the resulting algorithms are much higher in the case of AZ. The number of evaluated algorithms before achieving success ($\mathbb{S}_c$) for AZ is calculated only for the runs where it could find an acceptable algorithm. Even then, MOAZ was able to get an improvement of approximately 82% in $\mathbb{S}_c$. So, we can clearly see that MOAZ is consistently outperforming AZ in terms of both complexities of the resulting algorithms and success. One aspect to note here is that MOAZ does not compromise anything in terms of error, which is clear from the swarm plot represented in Figure 13. This plot shows the distributions of errors for all the algorithms obtained by both AZ and MOAZ. We can see that MOAZ has a wider distribution of errors, between 0 and 0.6, as these two values were set as the bounds on the error in MOAZ. In the case of AZ, the distribution is more concentrated toward zero error solutions, which is expected as AZ only focuses on reducing error. On the other hand, if we look

Table 1: Comparison of median complexity ($\mathbb{C}$) of the discovered algorithms, number of algorithms evaluated before success ($\mathbb{S}_c$), and success percentages ($\mathbb{S}_r$) for 30 runs of AZ and MOAZ framework. The standard deviations are also shown in parentheses.

Criterion	AZ	MOAZ
$\mathbb{C}$	79.5 (16.8)	**30 (2.17)**
$\mathbb{S}_c$	$8.9\times10^5(6.95\times10^5)$	$\mathbf{1.6\times10^5(1.8\times10^5)}$
$\mathbb{S}_r$	66.67	**100**

at the swarm plot for the complexity of the resulting algorithms in Figure 14, we can see that MOAZ algorithms are concentrated towards the lower end of the spectrum (between 20 and 60), whereas AZ algorithms are distributed between 60 and 140. These plots make it clear that MOAZ is able to provide consistent algorithms with the required error and complexity values.

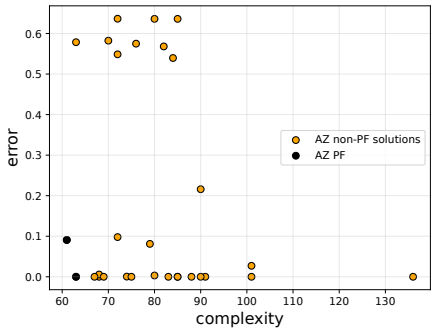

Figure 3: Combined results of AZ runs. Here PF refers to the Pareto Front of the combined results of AZ.

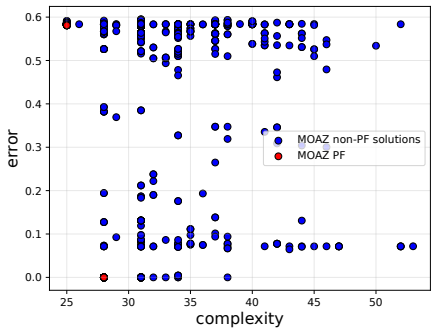

Figure 4: Combined results of MOAZ runs. Here PF refers to the Pareto Front of the combined results of AZ.

4.3.1 Search Speed. AZ uses multiple restarts to search for an algorithm that is sufficiently good in terms of the error to stop processing. If it was not able to find a working solution (defined

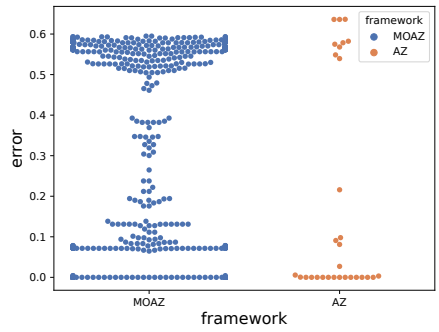

Figure 5: Swarm Plot comparison between AZ and MOAZ for error. In this plot, if the algorithms have equal errors, they are placed on the same line side-by-side.

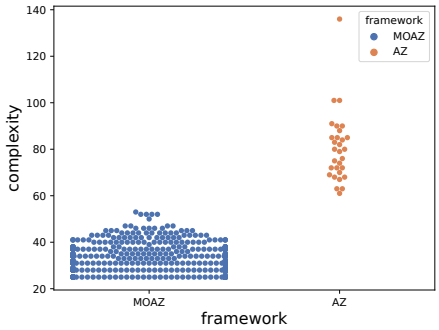

Figure 6: Swarm Plot comparison between AZ and MOAZ for complexity. In this plot, if the algorithms have equal complexity, they are placed on the same line side-by-side.

by a sufficient fitness measure) within a fixed number of algorithm evaluations, it stops and restarts the search with a different initial population. We were interested in determining the number of algorithm evaluations ($\mathbb{S}_c$) required to find the first solutions that meet the requirement of sufficient error. Clearly, a lower $\mathbb{S}_c$ is always preferred.

A comparison of the number of algorithm evaluations required before success for AZ and MOAZ over 30 runs is shown in Figure 7. It can be seen that the values for MOAZ are more concentrated towards the lower end of the spectrum. The figure does not contain runs of AZ for which it could not achieve any success. Again, MOAZ is showing more consistent results than AZ for the linear regression problem. MOAZ is able to converge or find acceptable solutions earlier than AZ. This phenomenon can be attributed mainly to the nature of the diversity maintained in the population of MOAZ. As it is trying to optimize two conflicting objectives, it is able to better preserve diversity in the population. AZ solutions are losing diversity faster because a locally optimal solution has the tendency to attract other solutions, and eventually, the population collapses. MOAZ is able to avoid this collapse to local optima by maintaining diversity because of its multi-objective character. For this reason,

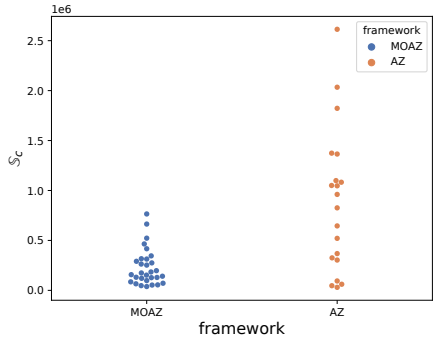

Figure 7: Comparison of the number of algorithm evaluations before success ($\mathbb{S}_c$) for AZ and MOAZ for the linear regression problem. MOAZ values are more concentrated towards the lower end of the spectrum. This figure does not include approximately 33% of the points for which AZ could not achieve success.

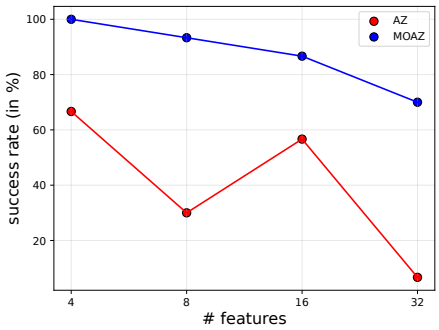

Figure 8: Rate of success ($\mathbb{S}_r$) for AZ and MOAZ for varying dimensions of features for linear regression.

MOAZ requires fewer restarts than AZ. A Wilcoxon Signed-Rank test revealed that the MOAZ results on error, algorithm complexity, and the number of evaluated algorithms to success were statistically significant compared to AZ results with p-values of 0.02, 1.9×10^{-9} and 0.002, respectively.

4.4 Scaling Experiment

So far, we have experimented with a fixed number of features d=4. To extend this to more dimensions, we have tested with d=8, 16, 32 for 30 runs of each of the frameworks. As we increase the number of dimensions for the linear regression problem, the search complexity becomes more difficult and we can expect to get less success rate for both AZ and MOAZ. How different is the performance of AZ and MOAZ as the number of dimensions is increased? We have compared the two frameworks in terms of the success rates, for multiple runs and for varying dimensionality. From Figure 8, we can see that the performance of AZ is quite arbitrary as the success rate is not always decreasing as the dimensions go up. For example, AZ got more success when the feature size is 16, as compared to when the size is 8. On the other hand, MOAZ is showing very consistent results. The number of convergences drops as the dimensions increase. At d=32, it is still able to get approximately 80% success rate, whereas AZ solutions achieve less than 20% success.

4.5 Non-linear Regression

While testing on the linear regression problem, we have seen that MOAZ is able to clearly outperform AZ in simple tasks. So, we wanted to extend the experiments to check if the performance holds for more complicated tasks like non-linear regression. The same non-linear regression tasks used in [29] are utilized for these experiments where the data is generated using a teacher neural network and the process needs to find the code for the teacher neural network[2]. The Pareto fronts for both AZ and MOAZ are displayed in Figure 9, showing that MOAZ is able to find a distribution of solutions within an $[0, 0.35]$ error range while reducing the complexity to some extent. The teacher algorithm has a complexity of 75 and in an ideal scenario, we should obtain a solution algorithm exactly matching that number. AZ found a close solution having a complexity of 107, while MOAZ found a close solution having a complexity of 95. MOAZ has a further advantage: It has found multiple approximations for the teacher network with some of them even having lesser complexities than the original neural network, trading off simplicity/complexity for error. This makes it a more powerful tool.

[2]Please refer to the supplementary materials for more information about the non-linear regression problem.

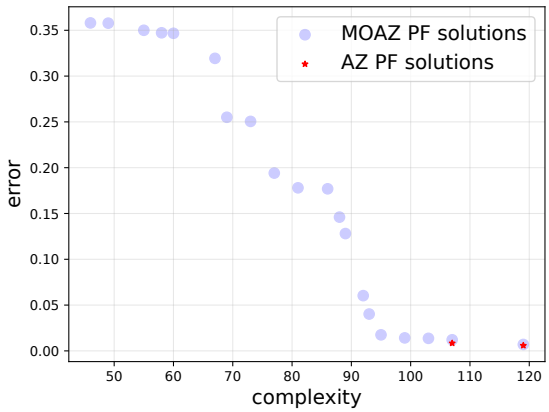

Figure 9: MOAZ and AZ PF for a non-linear problem under consideration.

5 CONCLUSION AND FUTURE WORK

In this paper, we have proposed a multi-objective version of the basic AutoML-Zero framework, which we called MOAZ. It has incorporated the capability of handling two conflicting objectives: the error rate of an algorithm and the corresponding complexity computed as a count of FLOPs. As MOAZ attempts to minimize the complexity of the algorithms, the final results are simpler and more

interpretable. These algorithms require less human intervention in terms of final processing (redundancy reduction) by offering progress toward the goal of AutoML: removing human designers from the loop for developing machine learning algorithms. Even though the current implementation of MOAZ uses the complexity of the algorithms as a secondary objective, it can be replaced with other objectives, such as deployability or energy efficiency.

For the linear regression problem, we have observed that MOAZ is able to perform better than AZ in two ways: smaller complexity of achieved algorithms with simultaneously reduced required number of evaluations before success. MOAZ is able to achieve an 87.4% reduction in the median complexity and 82% improvement in the number of evaluations required before success for a four-dimensional linear regression problem. Even when we have scaled up the dimensions of the linear regression problem, MOAZ has shown better and more consistent success achievement capability compared to AZ. Finally, we have also shown some preliminary results on a non-linear regression problem. Multiobjectivizing a problem, with a secondary but functionally helpful objective, has a clear advantage, as demonstrated in this paper. In the future, MOAZ can be applied to other more difficult problems, such as logistic regression, reinforcement learning, etc. MOAZ can also be extended to handle more objectives in the future.

SUPPLEMENTARY MATERIALS

6 FLOP COUNTS

We have approximated the Floating Point Operations (FLOP) count for each of 65 operations used in AutoML-Zero. These counts are listed in Table 2.

Table 2: The approximated number of FLOP for 65 operations used in AutoML-Zero. *VSize* and *MSize* refer to the dimensions of vectors and matrices used in the operations, respectively.

Operators	Approx. FLOP	Operators	Approx. FLOP
NO_OP	0	MATRIX_MAX_OP	*MSize*
SCALAR_SUM_OP	1	MATRIX_ABS_OP	*MSize*
SCALAR_DIFF_OP	1	MATRIX_HEAVYSIDE_OP	*MSize*
SCALAR_PRODUCT_OP	1	MATRIX_CONST_SET_OP	*MSize*
SCALAR_DIVISION_OP	1	SCALAR_VECTOR_PRODUCT_OP	*VSize*
SCALAR_MIN_OP	1	VECTOR_INNER_PRODUCT_OP	2*VSize
SCALAR_MAX_OP	1	VECTOR_OUTER_PRODUCT_OP	VSize*VSize
SCALAR_ABS_OP	1	SCALAR_MATRIX_PRODUCT_OP	MSize*MSize
SCALAR_HEAVYSIDE_OP	1	MATRIX_VECTOR_PRODUCT_OP	MSize*MSize
SCALAR_CONST_SET_OP	1	VECTOR_NORM_OP	2*VSize
SCALAR_SIN_OP	1	MATRIX_NORM_OP	MSize*MSize
SCALAR_COS_OP	1	MATRIX_TRANSPOSE_OP	*MSize*
SCALAR_TAN_OP	1	MATRIX_MATRIX_PRODUCT_OP	MSize*MSize
SCALAR_ARCSIN_OP	1	VECTOR_MEAN_OP	MSize*MSize
SCALAR_ARCCOS_OP	1	VECTOR_ST_DEV_OP	MSize*MSize
SCALAR_ARCTAN_OP	1	MATRIX_MEAN_OP	MSize*MSize
SCALAR_EXP_OP	1	MATRIX_ST_DEV_OP	MSize*MSize
SCALAR_LOG_OP	1	MATRIX_ROW_MEAN_OP	*MSize*
VECTOR_SUM_OP	*VSize*	MATRIX_ROW_ST_DEV_OP	*MSize*
VECTOR_DIFF_OP	*VSize*	SCALAR_GAUSSIAN_SET_OP	1
VECTOR_PRODUCT_OP	2*VSize	VECTOR_GAUSSIAN_SET_OP	*VSize*
VECTOR_DIVISION_OP	*VSize*	MATRIX_GAUSSIAN_SET_OP	MSize*MSize
VECTOR_MIN_OP	*VSize*	SCALAR_UNIFORM_SET_OP	1
VECTOR_MAX_OP	*VSize*	VECTOR_UNIFORM_SET_OP	*VSize*
VECTOR_ABS_OP	*VSize*	MATRIX_UNIFORM_SET_OP	MSize*MSize
VECTOR_HEAVYSIDE_OP	*VSize*	SCALAR_RECIPROCAL_OP	1
VECTOR_CONST_SET_OP	*VSize*	SCALAR_BROADCAST_OP	1
MATRIX_SUM_OP	*MSize*	VECTOR_RECIPROCAL_OP	*VSize*
MATRIX_DIFF_OP	*MSize*	MATRIX_RECIPROCAL_OP	MSize*MSize
MATRIX_PRODUCT_OP	MSize*MSize	MATRIX_ROW_NORM_OP	*MSize*
MATRIX_DIVISION_OP	*MSize*	MATRIX_COLUMN_NORM_OP	*MSize*
MATRIX_MIN_OP	*MSize*	VECTOR_COLUMN_BROADCAST_OP	*VSize*
VECTOR_ROW_BROADCAST_OP	*VSize*		

7 ADDITIONAL DESCRIPTION OF NON-LINEAR PROBLEM

In addition to the results mentioned in the main manuscript, here we provide some additional results that we obtained for the non-linear problem. The non-linear problem is formulated according to the process discussed in [29]. A teacher neural network acts as a non-linear regression model represented as $L(x_i) = u.ReLU(Mx_i)$ where M is a random 8×8 matrix and u is a random vector. In AZ/MOAZ nomenclature, this teacher neural network can be represented as shown in Figure 10. The task of AZ/MOAZ is to rediscover this neural network by getting signals from the dataset created by using the teacher neural network.

```
# sX/vX/mX: scalar/vector/matrix memory
# at address X.
def Setup():
 s2 = 0
def Predict(v0):
 v2 = dot(m0, v0)
 v3 = maximum(v2, v4)
 s1 = dot(v3, v1)
def Learn(v0, s0):
 s3 = s0 - s1
 s3 = s2 * s3
 v5 = s3 * v3
 v1 = v1 + v5
 v6 = s3 * v1
 v7 = heaviside(v2, 1.0)
 v6 = v7 * v6
 m1 = outer(v6, v0)
 m0 = m0 + m1
```

Figure 10: Illustration of the teacher neural network used as a non-linear regression model for generating labels for non-linear regression.

7.1 Experimental Setting

To perform the experiments with MOAZ, we used the following lower and upper bounds on complexity and error: the lower bound was 0 for both objectives, whereas the upper bound was 200 for complexity and 0.4 for error. We permit all instructions used by the teacher neural network in the search space. A target error for the problem (e_T) is defined as 5×10^{-2}. So, a run is considered to be successful if it could find at least one algorithm that has less than 5×10^{-2} error. Both AZ and MOAZ are run 30 times with different random seeds. The results and some comparisons of the performance of both frameworks are provided in Section 7.2.

7.2 Performance Comparison

The combined results for AZ and MOAZ are provided in Figure 11 and Figure 12, respectively. After combining the results for all 30 runs, the non-dominated solutions from all discovered algorithms are identified. These solutions are marked in different colors in the figures.

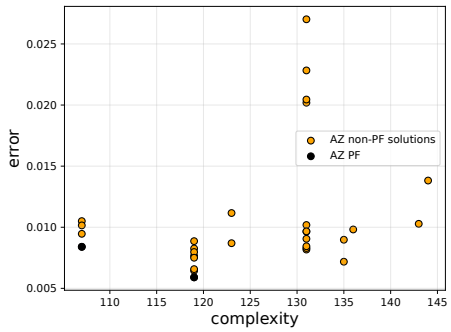

Figure 11: Combined results of AZ runs. Here PF refers to the Pareto Front of the combined results of AZ.

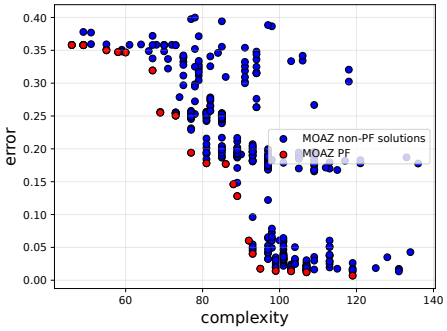

Figure 12: Combined results of MOAZ runs. Here PF refers to the Pareto Front of the combined results of AZ.

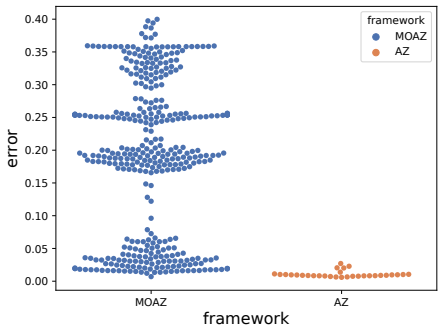

Figure 13: Swarm Plot comparison between AZ and MOAZ for error. In this plot, if the algorithms have equal errors, they are placed on the same line side-by-side.

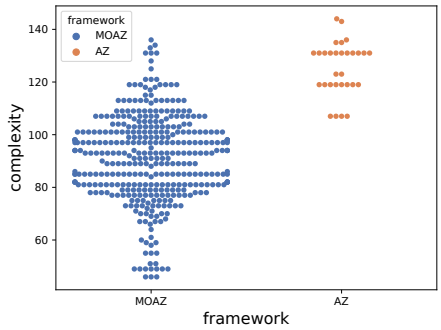

Figure 14: Swarm Plot comparison between AZ and MOAZ for complexity. In this plot, if the algorithms have equal complexity, they are placed on the same line side-by-side.

The distribution of errors and complexity is depicted in Figures Figure 13 and Figure 14, respectively. As the error range is $[0, 0.4]$, MOAZ has found a good distribution of solutions in the specified range. In contrast, most of the AZ solutions are concentrated toward the 0-error region. Similarly, for the complexity of solutions, MOAZ has found complexities in the range of $[50, 140]$ whereas all the solutions are beyond 100 complexity in the case of AZ. One of the interesting applications of finding a broad distribution is platform-based designs. Depending on different platforms, users might want to use different solutions. If the computational capability of a platform is on the lower side, the users can use a low-complexity model with some trade-off in accuracy. This is not possible with AZ solutions because AZ does not take into account complexity. The solutions of MOAZ are statistically significant compared to the AZ solutions both in terms of error and complexity with p-value in the order of 10^{-9}.

7.3 Discovered Algorithms

It is not possible to show all the algorithms discovered by both AZ and MOAZ. In this subsection, we are showing one representative algorithm for each of the frameworks. The AZ solution is shown in Figure 15 and the MOAZ solution is presented in Figure 16.

There are not many differences between these representative algorithms, apart from the initialization in the *Setup* component. MOAZ solution initializes just the two layers in the neural network but AZ solution initializes some other vectors as well. Please note that these are discovered algorithms with more than 99% accuracy. Some of these algorithms are very close to the teacher neural network. MOAZ also provides other solutions which are not this close to the teacher neural network but trade off some accuracy for the complexity.

REFERENCES

[1] Peter John Angeline. Genetic programming and emergent intelligence. *Advances in genetic programming*, 1:75–98, 1994.
[2] Wolfgang Banzhaf and William B. Langdon. Some considerations on the reason for bloat. *Genetic Programming and Evolvable Machines*, 3:81–91, 2002.
[3] Wolfgang Banzhaf, Frank D. Francone, Robert E. Keller, and Peter Nordin. *Genetic Programming — An Introduction: On the Automatic Evolution of Computer Programs and Its Applications.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1998. ISBN 155860510X.
[4] Markus Brameier and Wolfgang Banzhaf. *Linear Genetic Programming.* Springer, 2007.
[5] Tim Brys, Anna Harutyunyan, Peter Vrancx, Matthew E Taylor, Daniel Kudenko, and Ann Nowé. Multi-objectivization of reinforcement learning problems by reward shaping. In *2014 international joint conference on neural networks (IJCNN)*,

```
# sX/vX/mX: scalar/vector/matrix memory
# at address X.
def Setup():
 v8 = gaussian(2.24632, 0.0306014, n_features)
 m0 = gaussian(0.0295614, 0.0271724, (n_features,
n_features)) # 1st layer weights
 s3 = 0.0350772 # Learning rate
 v4 = gaussian(0.785895, 0.783264, n_features)
 v2 = gaussian(-0.39658, 0.93701, n_features) # 2nd layer
weights
def Predict(v0): # v0: features
 v3 = dot(m0, v0) # Apply 1st layer weights
 v4 = maximum(v3, v6) # Apply ReLU
 s1 = dot(v4, v2) # Apply 2nd layer weights
def Learn(v0, s0): # s0: labels, s1: predictions
 s4 = s0 - s1 # Compute error
 s4 = s3 * s4 # Scale by learning rate
 v6 = s4 * v8
 v2 = v2 + v6 # Update 2nd layer weights.
 v7 = s4 * v2
 v8 = heaviside(v3, 1.0) # ReLU gradient
 v7 = v8 * v7
 m1 = outer(v7, v0)
 m0 = m0 + m1 # Update 1st layer weights.
```

Figure 15: Illustration of a working algorithm for non-linear regression discovered by AZ with a complexity 107.

```
# sX/vX/mX: scalar/vector/matrix memory
# at address X.
def Setup():
 v2 = gaussian(-0.0635549, 0.51511, n_features) # 2nd
layer weights
 m0 = gaussian(0.0391599, 0.0467162, (n_features,
n_features)) # 1st layer weights
 s3 = 0.0237734 # Learning rate
def Predict(v0): # v0: features
 v3 = dot(m0, v0) # Apply 1st layer weights
 v4 = maximum(v3, v5) # Apply ReLU
 s1 = dot(v4, v2) # Apply 2nd layer weights
def Learn(v0, s0): # s0: labels, s1: predictions
 s4 = s0 - s1 # Compute error
 s4 = s3 * s4 # Scale by learning rate
 v6 = s4 * v4
 v2 = v2 + v6 # Update 2nd layer weights.
 v7 = s4 * v2
 v8 = heaviside(v3, 1.0) # ReLU gradient
 v7 = v8 * v7
 m1 = outer(v7, v0)
 m0 = m0 + m1 # Update 1st layer weights.
```

Figure 16: Illustration of a working algorithm for non-linear regression discovered by MOAZ with a complexity of 95.

pages 2315–2322. IEEE, 2014.

[6] Mauro Castelli, Leonardo Vanneschi, and Sara Silva. Semantic search-based genetic programming and the effect of intron deletion. *IEEE transactions on cybernetics*, 44(1):103–113, 2013.

[7] John D. Co-Reyes, Yingjie Miao, Daiyi Peng, Esteban Real, Sergey Levine, Quoc V. Le, Honglak Lee, and Aleksandra Faust. Evolving reinforcement learning algorithms. *CoRR*, abs/2101.03958, 2021. URL https://arxiv.org/abs/2101.03958.

[8] Kalyanmoy Deb. Multi-objective optimization. In *Search methodologies*, pages 403–449. Springer, 2014.

[9] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and TAMT Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197, 2002.

[10] Guozhu Dong and Huan Liu. *Feature engineering for machine learning and data analytics*. CRC Press, 2018.

[11] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. Neural architecture search: A survey. *The Journal of Machine Learning Research*, 20(1):1997–2017, 2019.

[12] Peter J Fleming, Robin C Purshouse, and Robert J Lygoe. Many-objective optimization: An engineering design perspective. In *International conference on evolutionary multi-criterion optimization*, pages 14–32. Springer, 2005.

[13] Pieter Gijsbers, Erin LeDell, Janek Thomas, Sébastien Poirier, Bernd Bischl, and Joaquin Vanschoren. An open source automl benchmark. *arXiv preprint arXiv:1907.00909*, 2019.

[14] Francesco Giuliari, Irtiza Hasan, Marco Cristani, and Fabio Galasso. Transformer networks for trajectory forecasting. In *2020 25th international conference on pattern recognition (ICPR)*, pages 10335–10342. IEEE, 2021.

[15] Walid A. Hanafy, Tergel Molom-Ochir, and Rohan Shenoy. Design considerations for energy-efficient inference on edge devices. In *Proceedings of the Twelfth ACM International Conference on Future Energy Systems*, e-Energy '21, page 302–308, New York, NY, USA, 2021. Association for Computing Machinery.

[16] Xin He, Kaiyong Zhao, and Xiaowen Chu. Automl: A survey of the state-of-the-art. *Knowledge-Based Systems*, 212:106622, 2021.

[17] Yihui He, Ji Lin, Zhijian Liu, Hanrui Wang, Li-Jia Li, and Song Han. Amc: Automl for model compression and acceleration on mobile devices. In *Proceedings of the European conference on computer vision (ECCV)*, pages 784–800, 2018.

[18] Frank Hutter, Lars Kotthoff, and Joaquin Vanschoren, editors. *Automated Machine Learning - Methods, Systems, Challenges*. Springer, 2019.

[19] Joshua D Knowles, Richard A Watson, and David W Corne. Reducing local optima in single-objective problems by multi-objectivization. In *International conference on evolutionary multi-criterion optimization*, pages 269–283. Springer, 2001.

[20] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA, 1992. ISBN 0262111705.

[21] Michael A. Lones. Evolving continuous optimisers from scratch. *Genetic Programming and Evolvable Machines*, 22(4):395–428, Dec 2021.

[22] Ishrat Zahan Mukti and Dipayan Biswas. Transfer learning based plant diseases detection using resnet50. In *2019 4th International conference on electrical information and communication technology (EICT)*, pages 1–6. IEEE, 2019.

[23] Mihai Oltean. Evolving Evolutionary Algorithms Using Linear Genetic Programming. *Evolutionary Computation*, 13(3):387–410, 09 2005.

[24] Laurent Pagnier and Michael Chertkov. Physics-informed graphical neural network for parameter & state estimations in power systems. *arXiv preprint arXiv:2102.06349*, 2021.

[25] Florian Pfisterer, Stefan Coors, Janek Thomas, and Bernd Bischl. Multi-objective automatic machine learning with autoxgboostmc. *arXiv preprint arXiv:1908.10796*, 2019.

[26] Rong Qu, Graham Kendall, and Nelishia Pillay. The general combinatorial optimization problem: Towards automated algorithm design. *IEEE Computational Intelligence Magazine*, 15(2):14–23, 2020.

[27] Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V Le. Regularized evolution for image classifier architecture search. In *Proceedings of the AAAI conference on artificial intelligence*, volume 33, pages 4780–4789, 2019.

[28] Esteban Real, Chen Liang, David So, and Quoc Le. AutoML-zero: Evolving machine learning algorithms from scratch. In Hal Daumé III and Aarti Singh, editors, *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pages 8007–8019. PMLR, 13–18 Jul 2020.

[29] Esteban Real, Chen Liang, David So, and Quoc Le. Automl-zero: Evolving machine learning algorithms from scratch. In *International Conference on Machine Learning*, pages 8007–8019. PMLR, 2020.

[30] Dong Song, Malcolm I Heywood, and A Nur Zincir-Heywood. A linear genetic programming approach to intrusion detection. In *Genetic and Evolutionary Computation—GECCO 2003: Genetic and Evolutionary Computation Conference Chicago, IL, USA, July 12–16, 2003 Proceedings, Part II*, pages 2325–2336. Springer, 2003.

[31] Terence Soule and Robert B Heckendorn. An analysis of the causes of code growth in genetic programming. *Genetic Programming and Evolvable Machines*, 3:283–309, 2002.

[32] Emma Strubell, Ananya Ganesh, and Andrew McCallum. Energy and policy considerations for deep learning in NLP. *CoRR*, abs/1906.02243, 2019. URL http://arxiv.org/abs/1906.02243.