AUTO: Supervised Learning With Full Model Search and Global Optimization

Justin Lovinger[a] and Iren Valova[a]

[a]Computer and Information Science Department, University of Massachusetts Dartmouth, North Dartmouth, MA, USA

ARTICLE HISTORY
Compiled May 1, 2023

Abstract
The AUTO algorithm is presented to incrementally build models and solve supervised learning problems. AUTO uses traditional derivative-free optimization, like genetic algorithms, to search the problem space of arbitrary functions. With reinforcement learning, AUTO learns actions to guide its search process and gradually improve performance.

A comparative analysis is presented exploring supervised learning performance and interpretability of AUTO models. Results indicate AUTO outperforms genetic programming and rivals multilayer perceptrons.

AUTO automatically builds models to closely match datasets without the limited search space of traditional supervised learning. With enough time and computational resources, AUTO can generate any function.

KEYWORDS
Mathematical Optimization; Supervised Learning; Genetic Programming; Machine Programming

1. Introduction

Genetic programming (GP) (Poli et al., 2008; Whitley, 1994; Xing et al., 2015) is a popular algorithm for the automatic generation of a function or program. Here, we use the general term, machine programming (MP), to refer to a class of algorithms that similarly generate a function or program. The infinite and complex problem domain of MP makes it an exceptionally difficult problem. Nevertheless, the ability to more efficiently develop programs makes MP a tempting problem to solve.

We present AUTO Universal Task Optimization (AUTO). AUTO uses derivative-free optimization, like genetic algorithms (GA) (Whitley, 1994), to create a function mapping a set of given inputs to a set of output values. A set of unit tests can evaluate the effectiveness of this function and return a objective value to improve later iterations. A novel technique that defines functions as a set of stacked partial function layers allows

CONTACT Justin Lovinger. Email: auto@justinlovinger.com

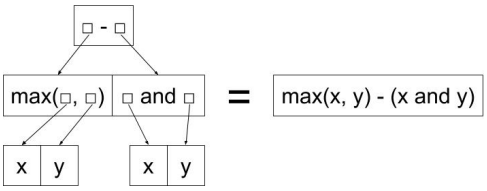

Figure 1. AUTO Solution to XOR

derivative-free optimization to explore the space of functions without resorting to graph based methods, as is popular in MP. A solution to the XOR problem, generated by AUTO, is depicted in Figure 1. The first layer contains a `-` operation, the second contains a `max` and `and` operation, and the final layer contains two `x` and two `y` terminals.

In all supervised learning (SL) problems, there is a theoretically optimal function f^* mapping a set of inputs $\boldsymbol{X}$ to a set of outputs $\boldsymbol{O}$. The goal of any SL algorithm is to find f^* using evidence from a dataset. Most SL models provide a function f_m with parameters that can be adjusted to learn a trained model function f'_m that is closer to f^*. This allows for fast training. However, for the vast majority of problems $f'_m \neq f^*$ for any initialization or training procedure, because the initial function f_m does not have the right form or parameters to ever equal f^*.

MP can generate arbitrary functions mapping inputs to outputs. As such, MP can perform SL (Espejo et al., 2009). Furthermore, MP can theoretically generate functions with equal or greater accuracy than any other SL model because MP can generate $f'_{mp} = f^*$ with optimal training, where f'_{mp} is a function generated by MP, even when $f'_{mp} = f'_m = f^*$ for another SL model f'_m. As in standard SL, outputs from a function generated by MP are compared to known target values, and error is fed back as an objective value to improve later iterations.

Unlike existing works, AUTO uses reinforcement learning (RL) to build programs and ML models. AUTO RL actions transform or extend a model, adding or changing operators, inputs, and parameters. The AUTO RL agent is like a human programmer refactoring code to improve performance and correctness.

The next section discusses the motivation for AUTO. Section 3 presents a discussion of related works. Section 4 details the AUTO algorithm. Section 5 provides a comparative analysis of AUTO on benchmark SL problems. Section 6 concludes.

2. Motivation

Machine programming (MP) techniques, like AUTO, can produce effective models compact enough for human interpretation. As such, AUTO is an effective tool for model-based fault diagnosis (Peng et al., 2020; W.-X. Yang, 2006; Zhang et al., 2005). AUTO can learn a model to predict faults in a real life system, such as rotating machinery. The model can then be examined to determine root causes, and the system can be improved to reduce future faults.

Unlike most machine learning techniques, AUTO does not always use all features or connect all features to all classes. AUTO can therefore produce compact models with fast activation times, using only a few comparisons in the best case.

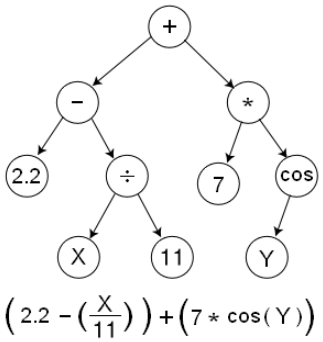

Figure 2. GP Tree (BAxelrod, n.d.)

With AUTO, we further the state-of-the-art in MP and ML, providing more effective models, better fault diagnosis, and more accurate prediction.

3. Related Works

MP is best known for the genetic programming (GP) method (Poli et al., 2008; Whitley, 1994; Xing et al., 2015). In brief, GP is a genetic algorithm (GA) (Deb et al., 2002; Lovinger et al., 2014; Uğuz, 2011; J. Yang & Honavar, 1998) acting on a graph, or tree, instead of a binary string. Figure 2 depicts a GP tree and its corresponding equation. Terminal leaf nodes are variables or constants. Non-leaf nodes are operators using terminals or the outputs of other operators. Every iteration, GP evaluates the fitness of each tree, and combines the most fit individuals to generate a new population of trees. Individuals can be combined with an analogy for GA crossover: a node in each parent tree is selected, and that node is replaced with the selected node and subtree of the other parent. Mutation can occur by selecting a node, and replacing that node with a randomly generated subtree.

In reinforcement learning (RL), agents take actions to maximize reward (Sutton & Barto, 2018). Unlike supervised learning, RL does not need labeled samples. Instead, RL agents train while acting by examining rewards associated with their actions. RL has been used to solve optimization problems (Chen et al., 2021; Mohamed et al., 2020; Zou et al., 2021).

4. Methodology

AUTO builds functions incrementally. Every iteration, a function f is modified by an action A^*, resulting in a new function. AUTO represents f as a series of stacked partial function layers. Each layer L_i has a number of symbols $s(L_i) \geq 1$, and requires a number of arguments $a(L_i) \geq 0$. AUTO builds f by substituting symbols from layer L_i into arguments of L_{i-1}. All layers $L_i \in L$ must satisfy

$$s(L_i) = a(L_{i-1}) \wedge a(L_i) = s(L_{i+1}) \tag{1}$$

where $a(L_0) = 1$ and $s(L_{|L|+1}) = 0$. That is to say, the first layer must have 1 symbol, and the last layer must have 0 arguments. Layer L_i is valid iff (1). Figure 3 depicts

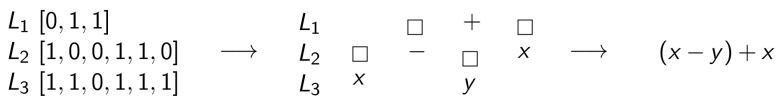

Figure 3. Building $f(x, y) = (x - y) + x$ from AUTO Layers

the process of building f from AUTO layers. Layers begin as encoded strings. Each layer string is decoded into a partial function. Each layer, except for the last, requires arguments from the next layer. Finally, symbols in each layer are substituted into the previous layer, forming f.

The best action for an iteration depends on f and an objective function F. Our AUTO algorithm implements 2 actions: add layer and replace layer. An action to delete the last layer was explored but found unnecessary alongside multiple tries.

The add layer action replaces the last layer L_n with a new layer L'_n, and adds an additional layer L'_{n+1}. L_n must be substituted because $a(L_n) = 0$ and $a(L'_n)$ must be > 0 to allow another layer. The add layer action allows exploration of larger programs.

The replace layer action selects a layer L_i, and replaces it with a new layer L'_i. The replace layer action escapes local minima that could otherwise result when L'_i is optimal only after more layers are added.

A special initialize action creates a layer with 1 terminal. Initialize is performed only once, on the first iteration.

Algorithm 1 presents the main AUTO loop. Multiple tries allow continued exploration of programs of a variety of sizes. AUTO retains reinforcement learning information across tries, improving performance for each subsequent try.

Algorithm 1 AUTO Algorithm

```
Given maximum number of tries n_t
Given maximum number of iterations per try n_i
Given optimization procedure O — See Sections 4.1, 4.3

f* ← best 1 layer function using O
loop
    f ← best 1 layer function using O
    loop
        A* ← best action — See Section 4.2
        f ← A*(f) using O
        if F(f) > F(f*) do
            f* ← f
        end if
    until n_i
until n_t
return f*
```

4.1. Optimizing Layers

Every action requires finding one or more replacement layers L'_i. Because L'_i must satisfy (1), we know $s(L'_i)$ and $a(L'_i)$. As such, L'_i can be encoded as a fixed length string. Without the need for variable length encoding, a vast variety of optimization algorithms can find L'_i, such as genetic algorithms (Deb et al., 2002; Lovinger et al., 2014; Uğuz, 2011; J. Yang & Honavar, 1998), particle swarm optimization (Kennedy, 2011), or hill climbing (Goldfeld et al., 1966; Mitchell et al., 1993; Xiao & Dunford, 2004).

The add layer action requires finding two layers, L'_i and L'_{i+1}, simultaneously. The selection of L'_i is highly dependent on L'_{i+1}, because the symbols in L'_{i+1} are arguments for L'_i. Recursive optimization can effectively search for the optimal combination of L'_i and L'_{i+1}. Recursive optimization depends on two optimizers. The first optimizer selects L'_i for evaluation. For every selected L'_i, the second optimizer searches for L'_{i+1} to maximize $F(f')$, where f' is f with substituted layers L'_i and L'_{i+1}. This process can be implemented as a decoding function for the first optimizer that decodes a binary string into L'_i and runs an optimizer that returns L'_{i+1}.

4.2. Selecting Actions

Selecting a random action A is trivial to implement. Although selecting a poor action A cannot decrease objective value $F(f^*)$, selecting the best action A^* optimally increases $F(f^*)$. Reinforcement learning (RL) allows AUTO to select A^*.

The RL agent can be simply specified as having a single state, a replace layer action for every layer, and an add layer action. Without continuous states or actions, or the need for delayed reward, a simple reward table is sufficient. The reward table R maps action to reward. Every iteration, $A^* = \text{argmax}_A R(A)$.

After every iteration, $R(A^*)$ is updated with a new reward r^*. Because our goal in applying A^* is to improve $F(f)$, $r^* = F(f) - F(f^{t-1})$, where f^{t-1} is the function from the previous iteration. Standard RL update rules adjust $R(A^*)$ towards r^*:

$$R(A^*) \leftarrow R(A^*) + \lambda(r^* - R(A^*)) \tag{2}$$

where $0 < \lambda \leq 1$ is the reward table update rate. (2) allows incrementally learning the average reward for each action.

When a new action A' is added to R, it must be given an initial reward value r_0. The choice of r_0 affects how frequently A' is selected before the true $R(A')$ is learned. We empirically select $r_0 = 0.5$. To encourage exploration, a reward growth parameter $r_g \geq 0$ can be selected. Every iteration, $R(A) \leftarrow R(A) + r_g \forall A \in R$. A small r_g allows the RL agent to eventually explore previously abandoned actions. We empirically select $r_g = 0.01$.

4.3. Decoding Binary Strings

Most derivative-free optimizers, like GA, represent parameters they optimize as binary strings. As such, we need to decode binary strings into lists of operators and terminals for AUTO layers.

Given a set of operators and terminals S, A simple method to select k items from S, is

to treat each sequence of b bits as an index, where b is the least number of bits required to represent $|S|-1$ and $|S|$ is the length of set S. Each sequence of b bits can be trivially converted into an integer index j, giving the j^{th} item of S. Repeating for each sequence of b bits provides a list of k items from a binary string of length $k*b$.

A sequence of individual indices can present problems when items must pass constraints, like (1). With constraints, many binary strings are invalid using the simple indices method. Constraints can be resolved by treating the entire bit string as a single index j to a list of k-permutations of set S, with repetition. A modified j^{th} k-permutation algorithm can find the j^{th} k-permutation passing an ordinal constraint function C using a binary search. The constraint function for the replace layer action $C_r(p_j) = a(p_j) - s(L_{i+1})$, where p_j is the j^{th} k-permutation represented as a layer. Note that we can trivially ensure $s(p_j) = a(L_{i-1})$ by setting $k = a(L_{i-1})$. The constraint function for the add layer action

$$C_a(p_j) = \begin{cases} -1 & a(p_j) = 0 \\ 0 & a(p_j) > 0 \end{cases} \tag{3}$$

Algorithm 2 presents the procedure for resolving C. Note that S must be sorted in ascending order of arguments, and terminals are treated as having 0 arguments.

Algorithm 2 Constrained i^{th} k-Permutation

```
if C(p_j) = 0 do
    return p_j
else if C(p_j) > 0 do
    return search(0, j − 1)
else — C(p_j) < 0
    return search(j + 1, |S| − 1)
end if
procedure search(j_l, j_u)
    let j′ = ⌊(j_l + j_u)/2⌋
    if C(p_j′) = 0 do
        return p_j′
    else if C(p_j′) > 0 do
        return search(j_l, j′ − 1)
    else — C(p_j′) < 0
        return search(j′ + 1, j_u)
    end if
end procedure
```

Although multiple indices can produce the same constrained k-permutation, binary search distributes these duplicates among the space of constrained k-permutations, minimising bias.

5. Supervised Learning Analysis and Comparison

Ground truth labels and targets make supervised learning an effective benchmark for machine programming and AUTO (D'Angelo et al., 2019; Hrnjica & Danandeh Mehr, 2018; Khanchi et al., 2018).

5.1. Datasets

Logical AND, OR, and XOR datasets provide easy to interpret results to demonstrate AUTO. All logical datasets are presented in their 2-dimensional variant with 1 regression target.

The well known iris dataset (Lichman, 2013) contains real value attributes describing various characteristics of a flower: sepal and petal width and length. The goal is to classify a flower into three types of iris. The iris dataset is low dimensional and relatively easy to classify.

California housing (CH) is a regression dataset (Huang et al., 2005, 2006; Pace, n.d.; Pace & Barry, 1997) predicting house value based on neighborhood and house statistics. Attributes are obtained using all block groups in California from the 1990 census.

The US postal service hand-written digit dataset (USPS) (Hull, 1994) contains 16x16 grey-scale images of hand written digits, gathered at the Center of Excellence in Document Analysis and Recognition (CEDAR) at SUNY Buffalo, as part of a project sponsored by the US postal service. Images are scanned from post office mail and contain a multitude of writers and styles. This high dimensional image dataset is significantly more difficult than iris.

The number of samples in each dataset, number of samples in the training set, attributes in each sample, and problem type for each benchmarked dataset is presented in Table 1. For classification datasets, the training set is given an even distribution of classes.

Table 1. Benchmark Datasets

Dataset	Type	Samples	Training Samples	Attributes	Classes / Outputs
AND	Regression	4	4	2	1
OR	Regression	4	4	2	1
XOR	Regression	4	4	2	1
Iris	Classification	150	90	4	3
CH	Regression	20640	500	8	1
USPS	Classification	11000	500	256	10

5.2. Models

Our configuration for each model used in this comparison is presented.

5.2.1. AUTO

AUTO runs for 10 tries and 25 iterations per try. It uses a genetic algorithm (GA) for layer optimization. The GA runs for 25 iterations, has a population of 12 chromosomes, a crossover chance of 0.7, a bit-flip mutation chance of 0.02, uses tournament selection with 2 competitors, and one-point crossover. The second optimizer, required for the add layer action, runs for 6 iterations and has a population of 3 chromosomes.

The objective function F for AUTO optimization is $F(f, \boldsymbol{X}, \boldsymbol{T}) = 1 - mae(f(\boldsymbol{X}), \boldsymbol{T})$, where f is the function learned by AUTO, $\boldsymbol{X}$ is an attribute matrix, $\boldsymbol{T}$ is a target matrix,

and *mae* is the mean absolute error function. Mean absolute error is used instead of mean squared error to avoid floating point overflow and because AUTO does not require a smooth gradient. Similar error functions, such as cross entropy, can take the place of *mae*.

The following operators are available for AUTO: $x + y$, $x - y$, $x * y$, x/y, x^y, $\max(x, y)$, and $x > y$, where x and y are arguments. $x/0 = 1$ to prevent division by zero errors. $x > y$ returns 1 for true and 0 for false. A selection of 100 constants, sampled at regular intervals from a beta distribution fitting training set attributes, is provided as terminals, in addition to a terminal for each attribute of an input vector.

For classification problems, labels are converted into one-hot target vectors $\vec{t}$, with a value of 1 for the component corresponding to the given label, and 0 for components corresponding to all other classes; vector length $|\vec{t}| = n_c$, where n_c is the number of classes. AUTO cannot directly output a vector because our set of operators for AUTO consists of only scalar functions. Instead, AUTO learns one function for each component of $\vec{t}$. The AUTO output $f(\vec{x})$ is formed by concatenating the output of each individual AUTO function $f_i(\vec{x})$, where $\vec{x}$ is an attribute vector.

5.2.2. Genetic Programming

Our genetic programming (GP) model runs for 250 iterations, has a population of 300 trees, a crossover chance of 0.5, mutation chance of 0.1, uses tournament selection with 2 competitors, one-point crossover, and a half-and-half scheme to generate the initial population. GP has access to the same operators and terminals as AUTO. As with AUTO, each label is converted into a one-hot vector, and GP learns one function for each component of the one-hot vector.

5.2.3. Multilayer Perceptron

Our multilayer perceptron (MLP) (Lovinger, 2018) model uses softplus, $f(x) = \ln(1+e^x)$, for hidden layers. ReLU and its softplus variant are effective in practice (Glorot et al., 2011; Maas et al., 2013; Tóth, 2013). Softmax output (Goodfellow et al., 2016) is used for classification datasets, and linear output is used for regression datasets. A BFGS optimizer (Broyden, 1970a; Broyden, 1970b; Dai, 2002; Fletcher, 1970; Goldfarb, 1970; Nocedal & Wright, 2006; Shanno, 1970), utilizing an approximation of second derivative information for improved performance, with Wolfe line search (Jiang & Jian, 2019; Nocedal & Wright, 2006; Yousif, 2020) and first-order-change initial step size (Nocedal & Wright, 2006) trains this model. Limited-memory BFGS (Bollapragada et al., 2018; Liu & Nocedal, 1989; Nocedal & Wright, 2006) trains this model on the USPS dataset due to a larger number of parameters. For each dataset, we start with 1 hidden neuron and increment by 1 until training error increases. The number of hidden neurons that minimizes training error is used. MLP uses 4 hidden neurons on Iris, 6 on CH, and 7 on USPS.

5.3. Results

Table 2 presents accuracy on training and testing sets for datasets in Section 5.1 and models in Section 5.2. Regression datasets present mean squared error in place of accuracy. All models achieve perfect accuracy on logical AND, OR, and XOR datasets.

Table 2. Supervised Learning Comparison

Dataset	Model	Train Acc/Err	Test Acc/Err
Iris	AUTO	100.00%	95.00%
	GP	82.22%	85.00%
	MLP	98.89%	95.00%
CH	AUTO	0.108	0.106
	GP	0.130	0.128
	MLP	0.056	0.127
USPS	AUTO	45.80%	37.06%
	GP	12.00%	11.43%
	MLP	99.20%	77.84%

AUTO proves its effectiveness, outperforming GP on all datasets, and matching or outperforming MLP on all datasets except USPS. High dimensional image recognition, as seen in USPS, proves difficult for both machine programming (MP) models, AUTO and GP. However, AUTO achieves over 3 times higher accuracy than GP.

AUTO is limited to 250 iterations across 10 tries for this experiment. However, as seen in Table 3, AUTO can continue to significantly improve its model, even close to this limit. More tries and iterations could yield further improvements.

Table 3. Tries and iterations when AUTO error changes on USPS class 3.

Try	Iteration	ξ
1	1	0.116
1	2	0.100
1	12	0.092
1	14	0.088
1	15	0.082
1	21	0.080
7	22	0.078
9	24	0.056
9	25	0.042

5.3.1. AUTO Functions

Table 4 examines functions learned by AUTO for datasets presented in Section 5.1. Number of symbols n_s in each function is presented, and functions of less than 25 symbols are displayed. Numeric terminals are rounded to 2 decimal places. AUTO function for class c is given by function f_c. Training error ξ for each function is also presented.

Even with the simple AND, OR, and XOR datasets, AUTO generates interesting solutions. Multiplication solves AND, the max operator solves OR, and XOR is solved by the max of inverted subtraction between arguments.

Functions for the Iris dataset demonstrate which classes are easy to predict, and which

Table 4. AUTO Functions

Dataset	n_s	Function	ξ
AND	3	$f_1(\vec{x}) = x_1 * x_2$	0
OR	3	$f_1(\vec{x}) = \max(x_2, x_1)$	0
XOR	7	$f_1(\vec{x}) = \max(x_2 - x_1, x_1 - x_2)$	0
Iris	3	$f_1(\vec{x}) = -0.55 > x_3$	0
	59	$f_2(\vec{x}) = ...$	0
	47	$f_3(\vec{x}) = ...$	0
CH	51	$f_1(\vec{x}) = ...$	0.108
USPS	15	$f_1(\vec{x}) = (x_{102} > -1.21) > (x_{120}^{1.7} - x_{115} * -0.93) * x_{255} + 3.56$	0.076
	11	$f_2(\vec{x}) = \max(-1.08, (x_{47} > x_{161}) * x_{130} + x_{31} > 2.36)$	0.076
	23	$f_3(\vec{x}) = -0.66 > \max(-0.71, \max(((x_{175} > x_{73}) > x_{91} - x_{55}) + x_{37}, -1.17 * x_{181} + (-0.83 * -0.8)^{2.96}))$	0.042
	13	$f_4(\vec{x}) = ((x_{176} > 0.27) * 0.87 * x_{171} - 0.1)^{-0.89} > x_{153}$	0.082
	97	$f_5(\vec{x}) = ...$	0.052
	19	$f_6(\vec{x}) = x_{116} - ((0.18^{0.87} + \max(x_{168}, x_{96})) - (x_{60} + x_{147} + x_{112})) > 2.06 - -1.06$	0.056
	49	$f_7(\vec{x}) = ...$	0.072
	35	$f_8(\vec{x}) = ...$	0.062
	15	$f_9(\vec{x}) = (-0.83 > x_{62})/x_{15}^{0.74} > (x_{238} + -1.08 > -1.14 + x_{124})$	0.088
	35	$f_{10}(\vec{x}) = ...$	0.076

are difficult. The first class is perfectly predicted with a single inequality, simply checking if pedal length is greater than -0.549. Note that attributes are normalized to a mean of 0 and standard deviation of 1. The second and third classes are also perfectly predicted, but require significantly larger functions. The second class is harder to predict than the third, requiring 12 more symbols.

As a regression dataset, CH requires many symbols to achieve low error. However, the 6th attribute is not used in the function generated by AUTO:
$((-0.965/(((((-0.710 > \max(-0.836, (((x_1 > -0.258)/(x_3 - x_3))^{x_4}))) + x_4)$
$*(-0.999 - (-0.597^{\max(-0.847,(x_4 * \max(x_4,(-0.226 * x_5))))}))^{-0.123})^{(-0.920 - -0.973) - -0.994}))$
$-(\max(x_7, (x_1 + \max(x_2, -0.799))) - x_7)) + x_8$

In the USPS dataset, we see a loose correlation between number of symbols and error. Functions 1, 2, 4, and 9 have few symbols and noticeably higher error than functions 5, 7, and 8, which have many symbols. Functions 3 and 6 are outliers with relatively few symbols and low error. This image recognition dataset requires complex functions for accurate prediction, but some classes are still easier to predict than other.

5.4. Performance Analysis

Each function f built by AUTO requires only as much computation as the sum of its operators. As such, time complexity is constant, $\mathcal{O}(1)$, with regard to the number of attributes in a dataset. AUTO uses only the attributes required to achieve high accuracy. AUTO has no upper bound on the number of operators in f, providing a theoretical time complexity upper bound of $\mathcal{O}(\infty)$. However, f is built incrementally, beginning

with a single terminal in one layer. This upper bound requires ∞ iterations to build.

The supervised learning objective function F, given in Section 5.2.1, requires calculating the mean squared error of f on a training set. Time complexity of every objective evaluation is $\mathcal{O}(n)$, where n is the number of samples in a given training set. Every AUTO iteration performs an action requiring a number of objective evaluations depending on the optimizer. The total number of evaluations depends on how many AUTO iterations are required to find f with sufficient objective value $F(f)$.

Our comparison AUTO model uses a genetic algorithm requiring 300 objective evaluations for the initialize and replace layer actions and 14400 evaluations for the add layer action. It runs for 250 AUTO iterations across 10 tries, requiring 75000 to 3600000 total evaluations, depending on actions taken.

6. Conclusion

By utilizing existing optimization algorithms to incrementally build a function, AUTO furthers the state-of-the-art in machine programming. Reinforcement learning allows AUTO to effectively mutate a function, adding depth and replacing operators and terminals. With this function mutation scheme, AUTO outperforms traditional genetic programming (GP).

With supervised learning as a benchmark, AUTO proves its ability to effectively generate functions. AUTO requires minimal configuration, solving problems independently, without user supervision or hyperparameter adjustment. AUTO functions are compact and sparse, providing excellent performance once training is complete.

AUTO scales exceptionally well with computational resources. AUTO can theoretically generate any function, limited only by available operators and terminals. With enough time, AUTO can perfectly solve any problem.

Functions generated by AUTO are interpretable, providing useful insights into the nature of a problem. We can learn by seeing selected operators, numbers of symbols, ignored attributes, and functions themselves. Imagine a machine learning expert called in to diagnose the high fault rate of an assembly line. An AUTO generated function could not only predict faults, but pinpoints factors leading to those faults, allowing engineers to fix root causes.

With only scalar operators and terminals, AUTO must divide its time between multiple functions to form a target vector. This limitation is especially pronounced on datasets with many classes, as seen with the USPS dataset in Section 5.3. AUTO can be extended with vector operators and terminals, directly outputting a target vector and focusing all iterations on improving performance.

AUTO is a machine programming platform. New optimizers and reinforcement learning models can improve performance. New actions can extend AUTO. New operators can solve different problem.

Disclosure Statement

The authors report there are no competing interests to declare.

References

BAxelrod. (n.d.). *A function represented as a tree structure*. https://en.wikipedia.org/wiki/Genetic_programming#/media/File:Genetic_Program_Tree.png

Bollapragada, R., Nocedal, J., Mudigere, D., Shi, H.-J., & Tang, P. T. P. (2018). A progressive batching L-BFGS method for machine learning. *International Conference on Machine Learning*, 620–629.

Broyden, C. G. (1970a). The convergence of a class of double-rank minimization algorithms 1. General considerations. *IMA Journal of Applied Mathematics*, *6*(1), 76–90.

Broyden, C. G. (1970b). The convergence of a class of double-rank minimization algorithms: 2. The new algorithm. *IMA Journal of Applied Mathematics*, *6*(3), 222–231.

Chen, J., Alnowibet, K., Annuk, A., & Mohamed, M. A. (2021). An effective distributed approach based machine learning for energy negotiation in networked microgrids. *Energy Strategy Reviews*, *38*, 100760. https://doi.org/https://doi.org/10.1016/j.esr.2021.100760

D'Angelo, G., Pilla, R., Tascini, C., & Rampone, S. (2019). A proposal for distinguishing between bacterial and viral meningitis using genetic programming and decision trees. *Soft Computing*, *23*(22), 11775–11791.

Dai, Y.-H. (2002). Convergence properties of the BFGS algoritm. *SIAM Journal on Optimization*, *13*(3), 693–701.

Deb, K., Pratap, A., Agarwal, S., & Meyarivan, T. (2002). A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation*, *6*(2), 182–197.

Espejo, P. G., Ventura, S., & Herrera, F. (2009). A survey on the application of genetic programming to classification. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, *40*(2), 121–144.

Fletcher, R. (1970). A new approach to variable metric algorithms. *The Computer Journal*, *13*(3), 317–322.

Glorot, X., Bordes, A., & Bengio, Y. (2011). Deep sparse rectifier neural networks. In G. Gordon, D. Dunson, & M. Dudík (Eds.), *Proceedings of the fourteenth international conference on artificial intelligence and statistics* (Vol. 15, pp. 315–323). PMLR. https://proceedings.mlr.press/v15/glorot11a.html

Goldfarb, D. (1970). A family of variable-metric methods derived by variational means. *Mathematics of Computation*, *24*(109), 23–26.

Goldfeld, S. M., Quandt, R. E., & Trotter, H. F. (1966). Maximization by quadratic hill-climbing. *Econometrica: Journal of the Econometric Society*, 541–551.

Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep learning*. MIT Press. http://www.deeplearningbook.org

Hrnjica, B., & Danandeh Mehr, A. (2018). *Optimized genetic programming applications: Emerging research and opportunities: Emerging research and opportunities*. IGI global.

Huang, G.-B., Chen, L., Siew, C. K., & others. (2006). Universal approximation using incremental constructive feedforward networks with random hidden nodes. *IEEE Trans. Neural Networks*, *17*(4), 879–892.

Huang, G.-B., Saratchandran, P., & Sundararajan, N. (2005). A generalized growing and pruning RBF (GGAP-RBF) neural network for function approximation. *IEEE Transactions on Neural Networks*, *16*(1), 57–67.

Hull, J. J. (1994). A database for handwritten text recognition research. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *16*(5), 550–554.

Jiang, X., & Jian, J. (2019). Improved fletcher–reeves and dai–yuan conjugate gradient methods with the strong wolfe line search. *Journal of Computational and Applied Mathematics*, *348*, 525–534.

Kennedy, J. (2011). Particle swarm optimization. In C. Sammut & G. I. Webb (Eds.), *Encyclopedia of machine learning* (pp. 760–766). Springer US. https://doi.org/10.1007/978-0-387-30164-8_630

Khanchi, S., Vahdat, A., Heywood, M. I., & Zincir-Heywood, A. N. (2018). On botnet detection with genetic programming under streaming data label budgets and class imbalance. *Swarm and Evolutionary Computation*, *39*, 123–140.

Lichman, M. (2013). *UCI machine learning repository*. University of California, Irvine, School of Information; Computer Sciences. http://archive.ics.uci.edu/ml

Liu, D. C., & Nocedal, J. (1989). On the limited memory BFGS method for large scale optimization. *Mathematical Programming*, *45*(1), 503–528.

Lovinger, J. (2018). *A tutorial on supervised learning from the perspective of mathematical optimization* [MS Thesis]. University of Massachusetts Dartmouth.

Lovinger, J., Valova, I., Rogers, M., Nadeau, R., & Gueorguieva, N. (2014). Harnessing mother nature: Optimizing genetic algorithms for adaptive systems. *Procedia Computer Science*, *36*, 523–528.

Maas, A. L., Hannun, A. Y., & Ng, A. Y. (2013). Rectifier nonlinearities improve neural network acoustic models. *Proc. ICML*, *30*(1), 3–9.

Mitchell, M., Holland, J., & Forrest, S. (1993). When will a genetic algorithm outperform hill climbing. In J. Cowan, G. Tesauro, & J. Alspector (Eds.), *Advances in neural information processing systems* (Vol. 6, pp. 51–58). Morgan-Kaufmann. https://proceedings.neurips.cc/paper/1993/file/ab88b15733f543179858600245108dd8-Paper.pdf

Mohamed, M. A., Jin, T., & Su, W. (2020). Multi-agent energy management of smart islands using primal-dual method of multipliers. *Energy*, *208*, 118306. https://doi.org/https://doi.org/10.1016/j.energy.2020.118306

Nocedal, J., & Wright, S. (2006). *Numerical optimization*. Springer Science & Business Media.

Pace, R. K. (n.d.). *California housing*. http://www.dcc.fc.up.pt/~ltorgo/Regression/cal_housing.html

Pace, R. K., & Barry, R. (1997). Sparse spatial autoregressions. *Statistics & Probability Letters*, *33*(3), 291–297.

Peng, B., Wan, S., Bi, Y., Xue, B., & Zhang, M. (2020). Automatic feature extraction and construction using genetic programming for rotating machinery fault diagnosis. *IEEE Transactions on Cybernetics*, *51*(10), 4909–4923.

Poli, R., Langdon, W. B., McPhee, N. F., & Koza, J. R. (2008). *A field guide to genetic programming*. Lulu. http://www.gp-field-guide.org.uk

Shanno, D. F. (1970). Conditioning of quasi-newton methods for function minimization. *Mathematics of Computation*, *24*(111), 647–656.

Sutton, R. S., & Barto, A. G. (2018). *Reinforcement learning: An introduction*. MIT press.

Tóth, L. (2013). Phone recognition with deep sparse rectifier neural networks. *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, 6985–6989. https://doi.org/10.1109/ICASSP.2013.6639016

Uğuz, H. (2011). A two-stage feature selection method for text categorization by using information gain, principal component analysis and genetic algorithm. *Knowledge-Based Systems*, *24*(7), 1024–1032.

Whitley, D. (1994). A genetic algorithm tutorial. *Statistics and Computing*, *4*(2), 65–85.

Xiao, W., & Dunford, W. G. (2004). A modified adaptive hill climbing MPPT method for photovoltaic power systems. *2004 IEEE 35th Annual Power Electronics Specialists Conference (IEEE Cat. No.04CH37551)*, *3*, 1957–1963. https://doi.org/10.1109/PESC.2004.1355417

Xing, W., Guo, R., Petakovic, E., & Goggins, S. (2015). Participation-based student final performance prediction model through interpretable genetic programming: Integrating learning analytics, educational data mining and theory. *Computers in Human Behavior*, *47*, 168–181.

Yang, J., & Honavar, V. (1998). Feature subset selection using a genetic algorithm. *IEEE Intelligent Systems and Their Applications*, *13*(2), 44–49.

Yang, W.-X. (2006). Establishment of the mathematical model for diagnosing the engine valve faults by genetic programming. *Journal of Sound and Vibration*, *293*(1-2), 213–226.

Yousif, O. O. O. (2020). The convergence properties of RMIL+ conjugate gradient method under the strong wolfe line search. *Applied Mathematics and Computation*, *367*, 124777.

Zhang, L., Jack, L. B., & Nandi, A. K. (2005). Fault detection using genetic programming. *Mechanical Systems and Signal Processing*, *19*(2), 271–289.

Zou, H., Tao, J., Elsayed, S. K., Elattar, E. E., Almalaq, A., & Mohamed, M. A. (2021). Stochastic multi-carrier energy management in the smart islands using reinforcement learning and unscented transform. *International Journal of Electrical Power & Energy Systems*, *130*, 106988. https://doi.org/https://doi.org/10.1016/j.ijepes.2021.106988