AutoML-Zero: Evolving Machine Learning Algorithms From Scratch

Esteban Real [*1] **Chen Liang** [*1] **David R. So** [1] **Quoc V. Le** [1]

Abstract

Machine learning research has advanced in multiple aspects, including model structures and learning methods. The effort to automate such research, known as AutoML, has also made significant progress. However, this progress has largely focused on the architecture of neural networks, where it has relied on sophisticated expert-designed layers as building blocks—or similarly restrictive search spaces. Our goal is to show that AutoML can go further: it is possible today to automatically discover complete machine learning algorithms just using basic mathematical operations as building blocks. We demonstrate this by introducing a novel framework that significantly reduces human bias through a generic search space. Despite the vastness of this space, evolutionary search can still discover two-layer neural networks trained by backpropagation. These simple neural networks can then be surpassed by evolving directly on tasks of interest, *e.g.* CIFAR-10 variants, where modern techniques emerge in the top algorithms, such as bilinear interactions, normalized gradients, and weight averaging. Moreover, evolution adapts algorithms to different task types: *e.g.*, dropout-like techniques appear when little data is available. We believe these preliminary successes in discovering machine learning algorithms from scratch indicate a promising new direction for the field.

1. Introduction

In recent years, neural networks have reached remarkable performance on key tasks and seen a fast increase in their popularity [*e.g.* He et al., 2015; Silver et al., 2016; Wu et al., 2016]. This success was only possible due to decades of machine learning (ML) research into many aspects of the field, ranging from learning strategies to new architectures [Rumelhart et al., 1986; LeCun et al., 1995; Hochreiter & Schmidhuber, 1997, among many others]. The length and difficulty of ML research prompted a new field, named *AutoML*, that aims to automate such progress by spending machine compute time instead of human research time (Fahlman & Lebiere, 1990; Hutter et al., 2011; Finn et al., 2017). This endeavor has been fruitful but, so far, modern studies have only employed constrained search spaces heavily reliant on human design. A common example is *architecture search*, which typically constrains the space by only employing sophisticated expert-designed layers as building blocks and by respecting the rules of backpropagation (Zoph & Le, 2016; Real et al., 2017; Tan et al., 2019). Other AutoML studies similarly have found ways to constrain their search spaces to isolated algorithmic aspects, such as the learning rule used during backpropagation (Andrychowicz et al., 2016; Ravi & Larochelle, 2017), the data augmentation (Cubuk et al., 2019a; Park et al., 2019) or the intrinsic curiosity reward in reinforcement learning (Alet et al., 2019); in these works, all other algorithmic aspects remain hand-designed. This approach may save compute time but has two drawbacks. First, human-designed components bias the search results in favor of human-designed algorithms, possibly reducing the innovation potential of AutoML. Innovation is also limited by having fewer options (Elsken et al., 2019b). Indeed, dominant aspects of performance are often left out (Yang et al., 2020). Second, constrained search spaces need to be carefully composed (Zoph et al., 2018; So et al., 2019; Negrinho et al., 2019), thus creating a new burden on researchers and undermining the purported objective of saving their time.

To address this, we propose to automatically search for *whole* ML algorithms using *little* restriction on form and *only* simple mathematical operations as building blocks. We call this approach *AutoML-Zero*, following the spirit of previous work which aims to learn with minimal human participation [*e.g.* Silver et al., 2017]. In other words, AutoML-Zero aims to search a fine-grained space simultaneously for the model, optimization procedure, initialization, and so on, permitting much less human-design and even allowing the discovery of non-neural network algorithms. To demonstrate that this is possible today, we present an initial solution to this challenge that creates algorithms competitive

[*]Equal contribution [1]Google Brain/Google Research, Mountain View, CA, USA. Correspondence to: Esteban Real <ereal@google.com>.

Proceedings of the 37^{th} International Conference on Machine Learning, Vienna, Austria, PMLR 119, 2020.

with backpropagation-trained neural networks.

The genericity of the AutoML-Zero space makes it more difficult to search than existing AutoML counterparts. Existing AutoML search spaces have been constructed to be dense with good solutions, thus deemphasizing the search method itself. For example, comparisons on the same space found that advanced techniques are often only marginally superior to simple random search (RS) (Li & Talwalkar, 2019; Elsken et al., 2019b; Negrinho et al., 2019). AutoML-Zero is different: the space is so generic that it ends up being quite sparse. The framework we propose represents ML algorithms as computer programs comprised of three *component functions*, `Setup`, `Predict`, and `Learn`, that performs initialization, prediction and learning. The instructions in these functions apply basic mathematical operations on a small memory. The operation and memory addresses used by each instruction are free parameters in the search space, as is the size of the component functions. While this reduces expert design, the consequent sparsity means that RS cannot make enough progress; *e.g.* good algorithms to learn even a trivial task can be as rare as 1 in 10^{12}. To overcome this difficulty, we use small proxy tasks and migration techniques to build highly-optimized open-source infrastructure capable of searching through 10,000 models/second/cpu core. In particular, we present a variant of functional equivalence checking that applies to ML algorithms. It prevents re-evaluating algorithms that have already been seen, even if they have different implementations, and results in a 4x speedup. More importantly, for better efficiency, we move away from RS.[1]

Perhaps surprisingly, evolutionary methods can find solutions in the AutoML-Zero search space despite its enormous size and sparsity. By randomly modifying the programs and periodically selecting the best performing ones on given tasks/datasets, we discover reasonable algorithms. We will first show that starting from empty programs and using data labeled by "teacher" neural networks with random weights, evolution can discover neural networks trained by gradient descent (Section 4.1). Next, we will minimize bias toward known algorithms by switching to binary classification tasks extracted from CIFAR-10 and allowing a larger set of possible operations. The result is evolved models that surpass the performance of a neural network trained with gradient descent by discovering interesting techniques like multiplicative interactions, normalized gradient and weight averaging (Section 4.2). Having shown that these ML algorithms are attainable from scratch, we will finally demonstrate that it is also possible to improve an existing algorithm by initializing the population with it. This way, evolution adapts the algorithm to the type of task provided. For example, dropout-like operations emerge when the task needs regularization and learning rate decay appears when the task requires faster convergence (Section 4.3). Additionally, we present ablation studies dissecting our method (Section 5) and baselines at various compute scales for comparisons by future work (Suppl. Section S10).

In summary, our contributions are:

- AutoML-Zero, the proposal to automatically search for ML algorithms from scratch with minimal human design;
- A novel framework with open-sourced code[1] and a search space that combines only basic mathematical operations;
- Detailed results to show potential through the discovery of nuanced ML algorithms using evolutionary search.

2. Related Work

AutoML has utilized a wide array of paradigms, including growing networks neuron-by-neuron (Stanley & Miikkulainen, 2002), hyperparameter optimization (Snoek et al., 2012; Loshchilov & Hutter, 2016; Jaderberg et al., 2017) and, neural architecture search (Zoph & Le, 2016; Real et al., 2017). As discussed in Section 1, AutoML has targeted many aspects of neural networks individually, using sophisticated coarse-grained building blocks. Mei et al. (2020), on the other hand, perform a fine-grained search over the convolutions of a neural network. Orthogonally, a few studies benefit from extending the search space to two such aspects simultaneously (Zela et al., 2018; Miikkulainen et al., 2019; Noy et al., 2019). In our work, we perform a fine-grained search over all aspects of the algorithm.

An important aspect of an ML algorithm is the optimization of its weights, which has been tackled by AutoML in the form of numerically discovered optimizers (Chalmers, 1991; Andrychowicz et al., 2016; Vanschoren, 2019). The output of these methods is a set of coefficients or a neural network that works well but is hard to interpret. These methods are sometimes described as "learning the learning algorithm". However, in our work, we understand *algorithm* more broadly, including the structure and initialization of the model, not just the optimizer. Additionally, our algorithm is not discovered numerically but *symbolically*. A symbolically discovered optimizer, like an equation or a computer program, can be easier to interpret or transfer. An early example of a symbolically discovered optimizer is that of Bengio et al. (1994), who optimize a local learning rule for a 4-neuron neural network using genetic programming (Holland, 1975; Forsyth et al., 1981; Koza & Koza, 1992). Our search method is similar but represents the program as a sequence of instructions. While they use the basic operations $\{+, -, \times, \div\}$, we allow many more, taking advantage of dense hardware computations. Risi & Stanley (2010) tackle the discovery of a biologically informed neural network learning rule too, but with a very different encoding.

[1] We open-source our code at https://github.com/google-research/google-research/tree/master/automl_zero#automl-zero

More recently, Bello et al. (2017) also search for a symbolic optimizer, but in a restricted search space of hand-tuned operations (*e.g.* "apply dropout with 30% probability", "clip at 0.00001", *etc.*). Our search space, on the other hand, aims to minimize restrictions and manual design. Unlike these three studies, we do not even assume the existence of a neural network or of gradients.

We note that our work also relates to program synthesis efforts. Early approaches have proposed to search for programs that improve themselves (Lenat, 1983; Schmidhuber, 1987). We share similar goals in searching for learning algorithms, but focus on common machine learning tasks and have dropped the self-reflexivity requirement. More recently, program synthesis has focused on solving problems like sorting (Graves et al., 2014), string manipulation (Gulwani et al., 2017; Balog et al., 2017), or structured data QA (Liang et al., 2016). Unlike these studies, we focus on synthesizing programs that solve the problem of *doing* ML.

Suppl. Section S1 contains additional related work.

3. Methods

AutoML-Zero concerns the automatic discovery of algorithms that perform well on a given set of ML tasks $\mathcal{T}$. First, *search experiments* explore a very large space of algorithms $\mathcal{A}$ for an optimal and generalizable $a^* \in \mathcal{A}$. The quality of the algorithms is measured on a subset $\mathcal{T}_{search} \subset \mathcal{T}$, with each search experiment producing a candidate algorithm. In this work, we apply random search as a baseline and evolutionary search as the main search method due to their simplicity and scalability. Once the search experiments are done, we select the best candidate by measuring their performances on another subset of tasks $\mathcal{T}_{select} \subset \mathcal{T}$ (analogous to standard ML model selection with a validation set). Unless otherwise stated, we use binary classification tasks extracted from CIFAR-10, a collection of tiny images each labeled with object classes (Krizhevsky & Hinton, 2009), and we calculate the average accuracy across a set of tasks to measure the quality of each algorithm. To lower compute costs and achieve higher throughput, we create small proxy tasks for $\mathcal{T}_{search}$ and $\mathcal{T}_{select}$ by using one random matrix for each task to project the input features to lower dimensionality. The projected dimensionality is $8 \leq F \leq 256$. Finally, we compare the best algorithm's performance against hand-designed baselines on the CIFAR-10 data in the original dimensionality (3072), holding out the CIFAR-10 test set for the final evaluation. To make sure the improvement is not specific to CIFAR-10, we further show the gain generalizes to other datasets: SVHN (Netzer et al., 2011), ImageNet (Chrabaszcz et al., 2017), and Fashion MNIST (Xiao et al., 2017). The *Experiment Details* paragraphs in Section 4 contain the specifics of the tasks. We now describe the search space and search method with sufficient detail to understand the results. For reproducibility, we provide the minutiae in the Supplement and the open-sourced code.

3.1. Search Space

We represent algorithms as computer programs that act on a small virtual memory with separate address spaces for scalar, vector and matrix variables (*e.g.* `s1`, `v1`, `m1`), all of which are floating-point and share the dimensionality of the task's input features (F). Programs are sequences of instructions. Each instruction has an operation—or *op*—that determines its function (*e.g.* "multiply a scalar with a vector"). To avoid biasing the choice of ops, we use a simple criterion: those that are typically learned by high-school level. We purposefully exclude machine learning concepts, matrix decompositions, and derivatives. Instructions have op-specific arguments too. These are typically addresses in the memory (*e.g.* "read the inputs from scalar address 0 and vector address 3; write the output to vector address 2"). Some ops also require real-valued constants (*e.g.* μ and σ for a random Gaussian sampling op), which are searched for as well. Suppl. Section S2 contains the full list of 65 ops.

```
# (Setup, Predict, Learn) = input ML algorithm.
# Dtrain / Dvalid = training / validation set.
# sX/vX/mX: scalar/vector/matrix var at address X.
def Evaluate(Setup, Predict, Learn, Dtrain,
Dvalid):
  # Zero-initialize all the variables (sX/vX/mX).
  initialize_memory()
  Setup() # Execute setup instructions.
  for (x, y) in Dtrain:
    v0 = x # x will now be accessible to Predict.
    Predict() # Execute prediction instructions.
    # s1 will now be used as the prediction.
    s1 = Normalize(s1) # Normalize the prediction.
    s0 = y # y will now be accessible to Learn.
    Learn() # Execute learning instructions.
  sum_loss = 0.0
  for (x, y) in Dvalid:
    v0 = x
    Predict() # Only Predict(), not Learn().
    s1 = Normalize(s1)
    sum_loss += Loss(y, s1)
  mean_loss = sum_loss / len(Dvalid)
  # Use validation loss to evaluate the algorithm.
  return mean_loss
```

Figure 1: Algorithm evaluation on one task. We represent an algorithm as a program with three component functions (`Setup`, `Predict`, `Learn`). These are evaluated by the pseudo-code above, producing a mean loss for each task. The search method then uses the median across tasks as an indication of the algorithm's quality.

Inspired by supervised learning work, we represent an algorithm as a program with three *component functions* that we call `Setup`, `Predict`, and `Learn` (*e.g.* Figure 5). The algorithm is evaluated as in Fig 1. There, the two for-loops implement the *training* and *validation phases*, processing

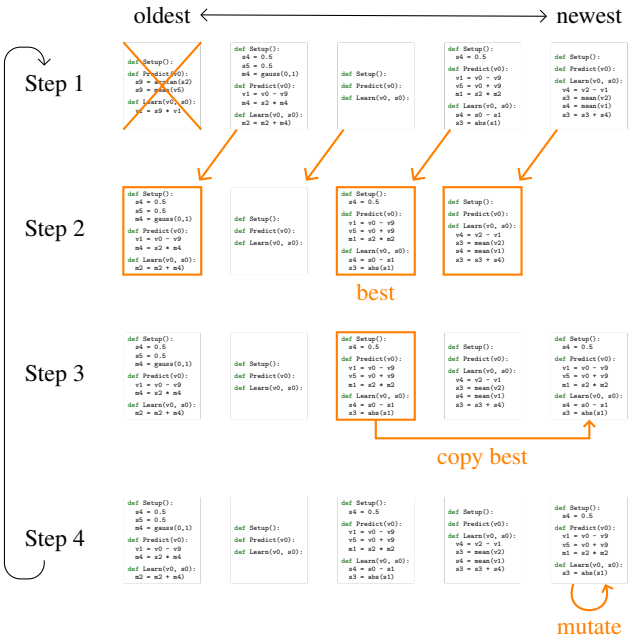


Figure 2: One cycle of the evolutionary method (Goldberg & Deb, 1991; Real et al., 2019). A population of P algorithms (here, P=5; laid out from left to right in the order they were discovered) undergoes many cycles like this one. First, we remove the oldest algorithm (step 1). Then, we choose a random subset of size T (here, T=3) and select the best of them (step 2). The best is copied (step 3) and mutated (step 4).

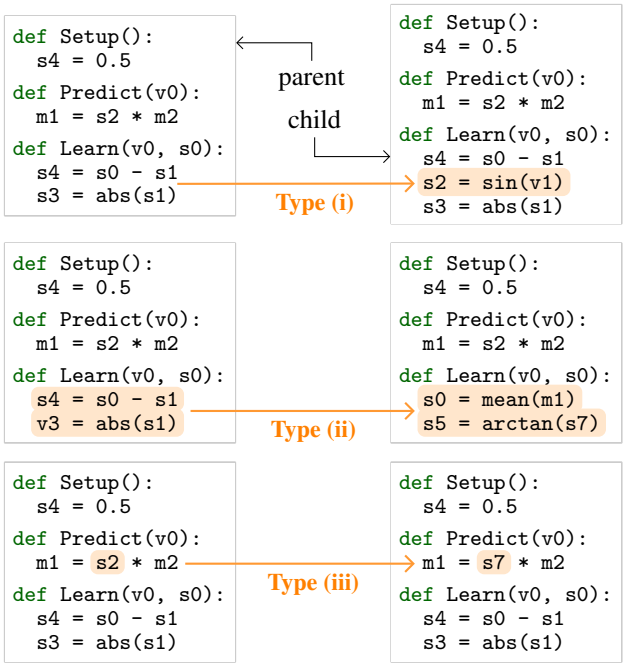


Figure 3: Mutation examples. Parent algorithm is on the left; child on the right. (i) Insert a random instruction (removal also possible). (ii) Randomize a component function. (iii) Modify an argument.

the task's examples one-at-a-time for simplicity. The training phase alternates `Predict` and `Learn` executions. Note that `Predict` just takes in the features of an example (*i.e.* `x`)—its label (*i.e.* `y`) is only seen by `Learn` afterward.

Then, the validation loop executes `Predict` over the validation examples. After each `Predict` execution, *whatever* value is in scalar address 1 (*i.e.* `s1`) is considered the prediction—`Predict` has no restrictions on what it can write there. For classification tasks, this prediction in $(-\infty, \infty)$ is normalized to a probability in $(0, 1)$ through a sigmoid (binary classification) or a softmax (multi-class). This is implemented as the `s1 = Normalize(s1)` instruction. The virtual memory is zero-initialized and persistent, and shared globally throughout the whole evaluation. This way, `Setup` can initialize memory variables (*e.g.* the weights), `Learn` can adjust them during training, and `Predict` can use them. This procedure yields an accuracy for each task. The median across D tasks is used as a measure of the algorithm's quality by the search method.

3.2. Search Method

Search experiments must discover algorithms by modifying the instructions in the component functions (`Setup`, `Predict`, and `Learn`; *e.g.* Figure 5). Unless otherwise stated, we use the *regularized evolution* search method because of its simplicity and recent success on architecture search benchmarks (Real et al., 2019; Ying et al., 2019; So et al., 2019). This method is illustrated in Figure 2. It keeps a population of P algorithms, all initially *empty*—*i.e.* none of the three component functions has any instructions/code lines. The population is then improved through cycles. Each cycle picks $T < P$ algorithms at random and selects the best performing one as the *parent*, *i.e. tournament selection* (Goldberg & Deb, 1991). This parent is then copied and *mutated* to produce a *child* algorithm that is added to the population, while the oldest algorithm in the population is removed. The mutations that produce the child from the parent must be tailored to the search space; we use a random choice among three types of actions: (i) insert a random instruction or remove an instruction at a random location in a component function, (ii) randomize all the instructions in a component function, or (iii) modify one of the arguments of an instruction by replacing it with a random choice (*e.g.* "swap the output address" or "change the value of a constant"). These are illustrated in Figure 3.

In order to reach a throughput of 2k–10k algorithms/second/cpu core, besides the use of small proxy tasks, we apply two additional upgrades: (1) We introduce a version of *functional equivalence checking* (FEC) that detects equivalent supervised ML algorithms, even if they have different implementations, achieving a 4x speedup. To do this, we record

the predictions of an algorithm after executing 10 training and 10 validation steps on a fixed set of examples. These are then truncated and hashed into a fingerprint for the algorithm to detect duplicates in the population and reuse previous evaluation scores. (2) We add hurdles (So et al., 2019) to reach further 5x throughput. In addition to (1) and (2), to attain higher speeds through parallelism, we distribute experiments across worker processes that exchange models through migration (Alba & Tomassini, 2002); each process has its own P-sized population and runs on a commodity CPU core. We denote the number of processes by W. Typically, $100<W<1000$ (we indicate the exact numbers with each experiment[2]). Workers periodically upload randomly selected algorithms to a central server. The server replies with algorithms randomly sampled across all workers, replacing half the local population (*i.e.* random *migration*). To additionally improve the quality of the search, we allow some workers to search on projected binary MNIST tasks, in addition to projected binary CIFAR-10, to promote diversity (see *e.g.* (Wang et al., 2019)). More details about these techniques can be found in Suppl. Section S3. Section 5 and Suppl. Section S9 contain ablation studies showing that all these techniques are beneficial.

For each experimental result, we include an *Experiment Details* paragraph with the exact values for meta-parameters like P and T. None of the meta-parameters were tuned in the final set of experiments at full compute scale. Most of them were either decided in smaller experiments (*e.g.* P), taken from previous work (*e.g.* T), or simply not tuned at all. In some cases, when uncertain about a parameter's appropriate value, we used a range of values instead (*e.g.* "$100 \leq P \leq 1000$"); different worker processes within the experiment use different values within the range.

Details: Generally, we use T=10, $100 \leq P \leq 1000$. Each child algorithm is mutated with probability U=0.9. Run time: 5 *days. Migration rate adjusted so that each worker process has fewer than 1 migration/s and at least 100 migrations throughout the expt. Specifics for each expt. in Suppl. Section S5. Suppl. Section S3 describes additional more general methods minutiae.*

4. Results

In the next three sections, we will perform experiments to answer the following three questions, respectively: "how difficult is searching the AutoML-Zero space?", "can we use our framework to discover reasonable algorithms with minimal human input?", and "can we discover different algorithms by varying the type of task we use during the search experiment?"

[2]The electricity consumption for our experiments (which were run in 2019) was matched with purchases of renewable energy.

4.1. Finding Simple Neural Nets in a Difficult Space

We now demonstrate the difficulty of the search space through random search (RS) experiments and we show that, nonetheless, interesting algorithms can be found, especially with evolutionary search. We will explore the benefits of evolution as we vary the task difficulty. We start by searching for algorithms to solve relatively easy problems, such as fitting linear regression data. Note that without the following simplifications, RS would not be able to find solutions.

Experiment Details: we generate simple regression tasks with 1000 *training and* 100 *validation examples with random 8-dim. feature vectors $\{x_i\}$ and scalar labels $\{L(x_i)\}$. L is fixed for each task but varies between them. To get affine tasks, $L(x_i) = u \cdot x_i + a$, where u and a are a random vector and scalar. For linear tasks, a=0. All random numbers were Gaussian (μ=0, σ=1). Evaluations use RMS error and the* `Normalize()` *instruction in Figure 1 is the identity. We restrict the search space by only using necessary ops and fixing component function lengths to those of known solutions. E.g., for a linear dataset,* `Learn` *has 4 instructions because linear SGD requires 4 instructions. To keep lengths fixed, insert/remove-instruction mutations are not allowed and component functions are initialized randomly. RS generates programs where all instructions are random (see Section 3.2) and selects the best at the end. Evolution expts. are small (W=1; D=* 3*;* 10k *algs./expt.); We repeat expts. until statistical significance is achieved. Full configs. in Suppl. Section S5. Note that the restrictions above apply *only* to this section (4.1).*

We quantify a task's difficulty by running a long RS experiment. We count the number of *acceptable algorithms*, *i.e.* those with lower mean RMS error than a hand-designed reference (*e.g.* linear regressor or neural network). The ratio of acceptable algorithms to the total number of algorithms evaluated gives us an *RS success rate*. It can also be interpreted as an estimate of the "density of acceptable algorithms" in the search space. We use this density as a measure of problem difficulty. For example, in the linear regression case, we looked for all algorithms that do better than a linear regressor with gradient descent. Even in this trivial task type, we found only 1 acceptable algorithm every 10^7, so we define 10^7 to be the difficulty of the linear regression task. We then run the evolution experiments with the same combined total number of evaluations as for RS. We measure the ratio of acceptable algorithms to the total number of algorithms evaluated, to get an *evolution success rate*. However, we only count at most 1 acceptable algorithm from each experiment; this biases the results *against* evolution but is necessary because a single experiment may yield multiple copies of a single acceptable algorithm. Even in the simple case of linear regression, we find that evolution is 5 times more efficient than RS. This stands in contrast to many previous AutoML studies, where the solutions are dense enough that RS can be competitive (Section 1).

Figure 4 summarizes the result of this analysis for 4 task types: the discovery of a full-algorithm/only-the-learning

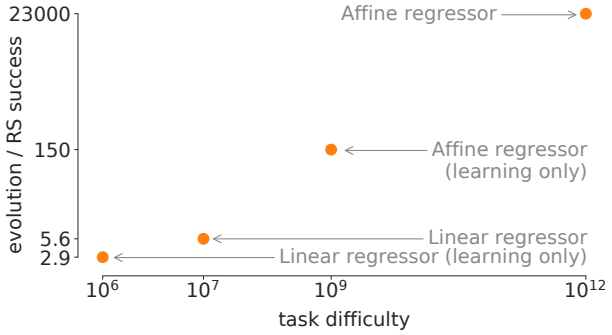


Figure 4: Relative success rate of evolution and random search (RS). Each point represents a different task type and the x-axis measures its difficulty (defined in the main text). As the task type becomes more difficult, evolution vastly outperforms RS, illustrating the complexity of AutoML-Zero when compared to more traditional AutoML spaces.

```
# sX/vX/mX = scalar/vector/matrix at address X.
# "gaussian" produces Gaussian IID random numbers.
def Setup():
  # Initialize variables.
  m1 = gaussian(-1e-10, 9e-09) # 1st layer weights
  s3 = 4.1 # Set learning rate
  v4 = gaussian(-0.033, 0.01) # 2nd layer weights
def Predict():  # v0=features
  v6 = dot(m1, v0) # Apply 1st layer weights
  v7 = maximum(0, v6) # Apply ReLU
  s1 = dot(v7, v4) # Compute prediction
def Learn():  # s0=label
  v3 = heaviside(v6, 1.0) # ReLU gradient
  s1 = s0 - s1 # Compute error
  s2 = s1 * s3 # Scale by learning rate
  v2 = s2 * v3 # Approx. 2nd layer weight delta
  v3 = v2 * v4 # Gradient w.r.t. activations
  m0 = outer(v3, v0) # 1st layer weight delta
  m1 = m1 + m0 # Update 1st layer weights
  v4 = v2 + v4 # Update 2nd layer weights
```

Figure 5: Rediscovered neural network algorithm. It implements backpropagation by gradient descent. Comments added manually.

for linear/affine regression data. The AutoML-Zero search space is generic but this comes at a cost: even for easy problems, good algorithms are sparse. As the problem becomes more difficult, the solutions become vastly more sparse and evolution greatly outperforms RS.

As soon as we advance to nonlinear data, the gap widens and we can no longer find solutions with RS. To make sure a good solution exists, we generate regression tasks using *teacher* neural networks and then verify that evolution can rediscover the teacher's code.

Experiment Details: tasks as above but the labeling function is now a teacher network: $L(x_i) = u \cdot ReLU(Mx_i)$, where M is a random 8×8 matrix, u is a random vector. Number of training examples up to 100k. *Single expt. Same search space restrictions as above, but now allowing ops used in 2-layer fully connected neural nets. After searching, we select the algorithm with the smallest RMS loss. Full configs. in Suppl. Section S5. Note that the restrictions above apply *only* to this section (4.1).*

When the search method uses only 1 task in $\mathcal{T}_{search}$ (*i.e.* $D=1$), the algorithm evolves the exact prediction function used by the teacher and hard-codes its weights. The results become more surprising as we increase the number of tasks in $\mathcal{T}_{search}$ (*e.g.* to $D=100$), as now the algorithm must find different weights for each task. In this case, evolution not only discovers the forward pass, but also "invents" backpropagation code to learn the weights (Figure 5). Despite its difficulty, we conclude that searching the AutoML-Zero space seems feasible and we should use evolutionary search instead of RS for more complex tasks.

4.2. Searching with Minimal Human Input

Teacher datasets and carefully chosen ops bias the results in favor of known algorithms, so in this section we replace them with more generic options. We now search among a long list of ops selected based on the simplicity criterion described in Section 3.1. The increase in ops makes the search more difficult but allows the discovery of solutions other than neural networks. For more realistic datasets, we use binary classification tasks extracted from CIFAR-10 and MNIST.

Experiment Details: We extract tasks from the CIFAR-10 and MNIST training sets; each of the datasets are searched on by half of the processes. For both datasets, the 45 pairs of the 10 classes yield tasks with 8000 *train/*2000 *valid examples.* 36 *pairs are randomly selected to constitute $\mathcal{T}_{search}$, i.e. search tasks;* 9 *pairs are held out for $\mathcal{T}_{select}$, i.e tasks for model selection. The CIFAR-10 test set is reserved for final evaluation to report results. Features are projected to $8 \leq F \leq 256$ dim. Each evaluation is on $1 \leq D \leq 10$ tasks. $W=10$k. From now on, we use the full setup described in Section 3.2. In particular, we allow variable component function length. Number of possible ops:* 7/58/58 *for* `Setup`/`Predict`/`Learn`*, resp. Full config. in Suppl. Section S5.*

Figure 6 shows the progress of an experiment. It starts with a population of empty programs and automatically invents improvements, several of which are highlighted in the plot. These intermediate discoveries are stepping stones available to evolution and they explain why evolution outperforms RS in this space. Each experiment produces a candidate algorithm using $\mathcal{T}_{search}$. We then evaluate these algorithms on unseen pairs of classes ($\mathcal{T}_{select}$) and compare the results to a hand-designed reference, a 2-layer fully connected neural network trained by gradient descent. The candidate algorithms perform better in 13 out of 20 experiments. To make sure the improvement is not specific to the small proxy tasks, we select the best algorithm for a final evaluation on binary classification with the original CIFAR-10 data.

Since we are evaluating on tasks with different dimensional-

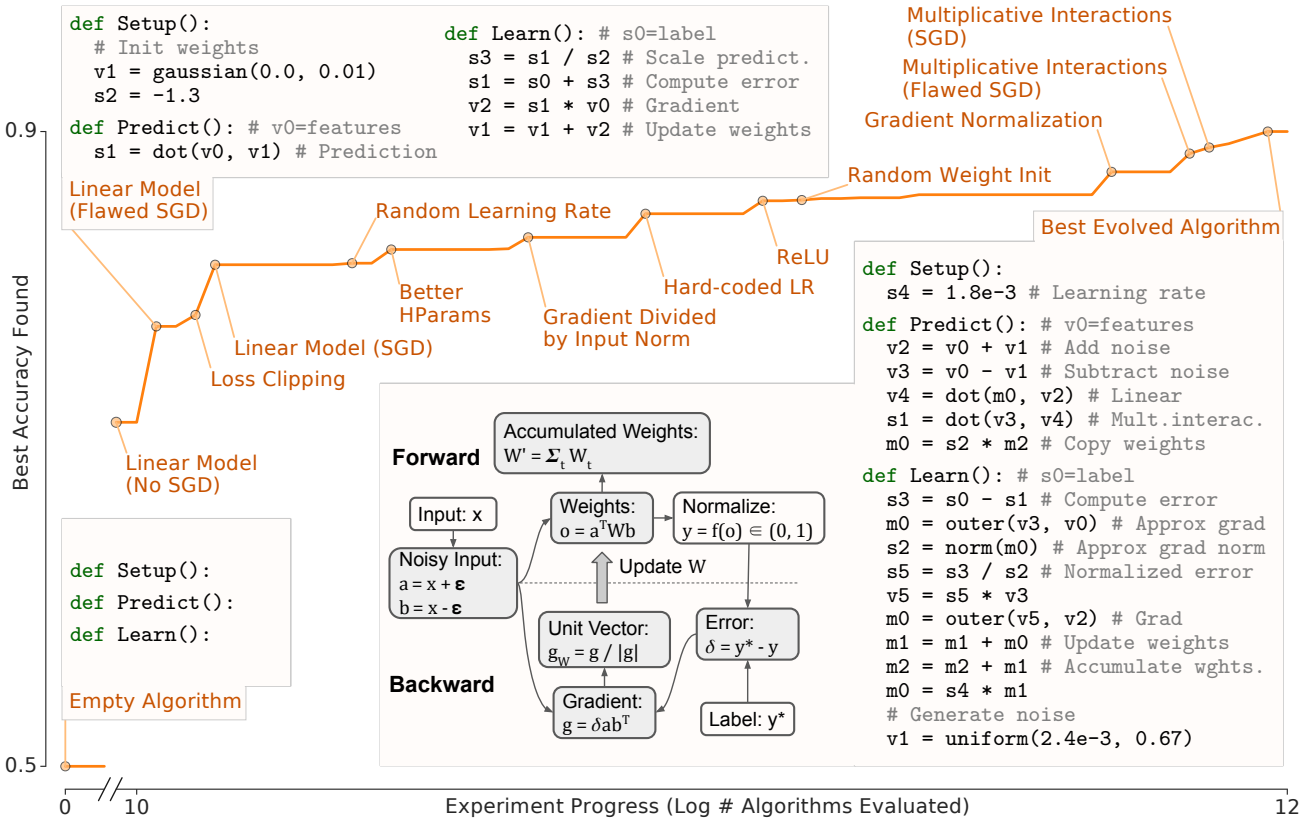


Figure 6: Progress of one evolution experiment on projected binary CIFAR-10. Callouts indicate some beneficial discoveries. We also print the code for the initial, an intermediate, and the final algorithm. The last is explained in the flow diagram. It outperforms a simple fully connected neural network on held-out test data and transfers to features 10x its size. Code notation is the same as in Figure 5. The x-axis gap is due to infrequent recording due to disk throughput limitations.

ity in the final evaluation, we treat all the constants in the best evolved algorithm as hyperparameters and tune them jointly through RS using the validation set. For comparison, we tune two hand-designed baselines, one linear and one nonlinear, using the same total compute that went into discovering and tuning the evolved algorithm. We finally evaluate them all on unseen CIFAR-10 test data. Evaluating with 5 different random seeds, the best evolved algorithm's accuracy ($84.06 \pm 0.10\%$) significantly outperforms the linear baseline (logistic regression, $77.65 \pm 0.22\%$) and the nonlinear baseline (2-layer fully connected neural network, $82.22 \pm 0.17\%$). This gain also generalizes to binary classification tasks extracted from other datasets: SVHN (Netzer et al., 2011) (88.12% for the best evolved algorithm *vs.* 59.58% for the linear baseline *vs.* 85.14% for the nonlinear baseline), downsampled ImageNet (Chrabaszcz et al., 2017) (80.78% *vs.* 76.44% *vs.* 78.44%), Fashion MNIST (Xiao et al., 2017) (98.60% *vs.* 97.90% *vs.* 98.21%). This algorithm is limited by our simple search space, which cannot currently represent some techniques that are crucial in state-of-the-art models, like batch normalization or convolution. Nevertheless, the algorithm shows interesting characteristics, which we describe below.

As a case study, we delve into the best algorithm, shown in Figure 6. The code has been cleaned for readability; we removed and rearranged instructions when this caused no difference in performance (raw code in Suppl. Section S6). The algorithm has the following notable features, whose usefulness we verified through ablations (more details in Suppl. Section S8): (1) Noise is added to the input, which, we suspect, acts as a regularizer:

$$\mathbf{a} = \mathbf{x} + \mathbf{u}; \mathbf{b} = \mathbf{x} - \mathbf{u}; \mathbf{u} \sim \mathbf{U}(\alpha, \beta)$$

where $\mathbf{x}$ is the input, $\mathbf{u}$ is a random vector drawn from a uniform distribution. (2) Multiplicative interactions (Jayakumar et al., 2020) emerge in a *bilinear* form: $\mathbf{o} = \mathbf{a}^\intercal\mathbf{W}\mathbf{b}$, where $\mathbf{o}$ is the output, and $\mathbf{W}$ is the weight matrix. (3) The gradient $\mathbf{g}$ w.r.t. the weight matrix $\mathbf{W}$ is computed correctly and is then normalized to be a unit vector:

$$\mathbf{g_w} = \frac{\mathbf{g}}{|\mathbf{g}|}; \mathbf{g} = \delta\mathbf{a}\mathbf{b}^\intercal; \delta = \mathbf{y}^* - \mathbf{y};$$

where δ is the error, y is the predicted probability, and y^* is the label. Normalizing gradients is a common heuris-

tic in non-convex optimization (Hazan et al., 2015; Levy, 2016). (4) The weight matrix $\mathbf{W}'$ used during inference is the accumulation of all the weight matrices $\{\mathbf{W_t}\}$ after each training step $\mathbf{t}$, *i.e.*: $\mathbf{W}' = \sum_t \mathbf{W_t}$. This is reminiscent of the *averaged perceptron* (Collins, 2002) and neural network *weight averaging* during training (Polyak & Juditsky, 1992; Goodfellow et al., 2016). Unlike these studies, the evolved algorithm *accumulates* instead of averaging, but this difference has no effect when measuring the accuracy of classification tasks (it does not change the prediction). As in those techniques, different weights are used at training and validation time. The evolved algorithm achieves this by setting the weights $\mathbf{W}$ equal to $\mathbf{W}'$ at the end of the `Predict` component function and resetting them to $\mathbf{W_t}$ right after that, at the beginning of the `Learn` component function. This has no effect during training, when `Predict` and `Learn` alternate in execution. However, during validation, `Learn` is never called and `Predict` is executed repeatedly, causing $\mathbf{W}$ to remain as $\mathbf{W}'$.

In conclusion, even though the task used during search is simple, the results show that our framework can discover commonly used algorithms from scratch.

4.3. Discovering Algorithm Adaptations

In this section, we will show wider applicability by searching on three different task types. Each task type will impose its own challenge (*e.g.* "too little data"). We will show that evolution specifically adapts the algorithms to meet the challenges. Since we already reached reasonable models from scratch above, now we save time by simply initializing the populations with the working neural network of Figure 5.

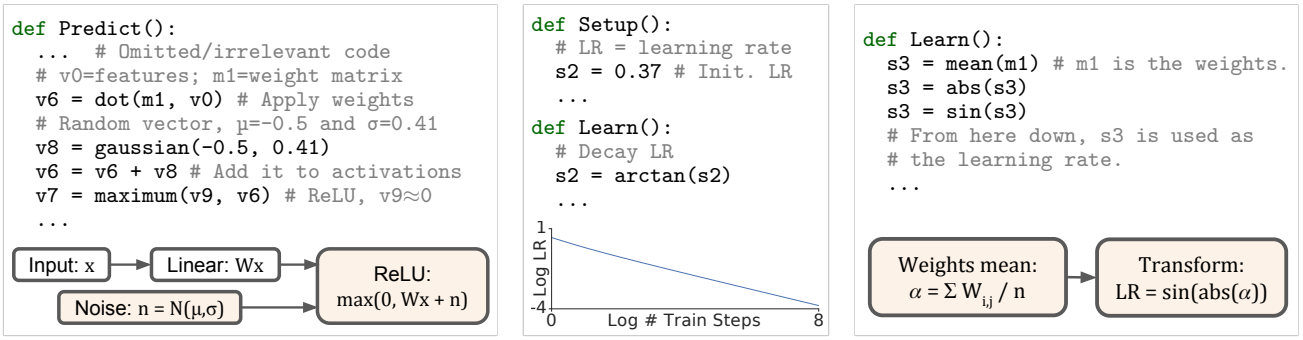
Experiment Details: The basic expt. configuration and datasets (binary CIFAR-10) are as in Section 4.2, with the following exceptions: $W=1k$; $F=16$; $10 \leq D \leq 100$; critical alterations to the data are explained in each task type below. Full configs. in Suppl. Section S5.

Few training examples. We use only 80 of the training examples and repeat them for 100 epochs. Under these conditions, algorithms evolve an adaptation that augments the data through the injection of noise (Figure 7a). This is referred to in the literature as a *noisy ReLU* (Nair & Hinton, 2010; Bengio et al., 2013) and is reminiscent of Dropout (Srivastava et al., 2014). Was this adaptation a result of the small number of examples or did we simply get lucky? To answer this, we perform 30 repeats of this experiment and of a control experiment. The control has 800 examples/100 epochs. We find that the noisy ReLU is reproducible and arises preferentially in the case of little data (expt: $8/30$, control: $0/30$, $p<0.0005$).

Fast training. Training on 800 examples/10 epochs leads to the repeated emergence of learning-rate decay, a well-known strategy for the timely training of an ML model (Bengio, 2012). An example can be seen in Figure 7b. As a control, we increase the number of epochs to 100. With overwhelming confidence, the decay appears much more often in the cases with fewer training steps (expt: $30/30$, control: $3/30$, $p<10^{-14}$).

Multiple classes. When we use all 10 classes of the CIFAR-10 dataset, evolved algorithms tend to use the transformed mean of the weight matrix as the learning rate (Figure 7c). (Note that to support multiple classes, labels and outputs are now vectors, not scalars.) While we do not know the reason, the preference is statistically significant (expt: $24/30$, control: $0/30$, $p<10^{-11}$).

Altogether, these experiments show that the resulting algorithms seem to adapt well to the different types of tasks.



(a) Adaptation to few examples. (b) Adaptation to fast training. (c) Adaptation to multiple classes.

Figure 7: Adaptations to different task types. (a) When few examples are available, evolution creates a noisy ReLU. (b) When fast training is needed, we get a learning rate decay schedule implemented as an iterated arctan map (top) that is nearly exponential (bottom). (c) With multiple classes, the mean of the weight matrix is transformed and then used as the learning rate. Same notation as in Figure 5; full algorithms in Suppl. Section S6.

5. Conclusion and Discussion

In this paper, we proposed an ambitious goal for AutoML: the automatic discovery of whole ML algorithms from basic operations with minimal restrictions on form. The objective was to reduce human bias in the search space, in the hope that this will eventually lead to new ML concepts. As a start, we demonstrated the potential of this research direction by constructing a novel framework that represents an ML algorithm as a computer program comprised of three component functions (`Setup`, `Predict`, `Learn`). Starting from empty component functions and using only basic mathematical operations, we evolved neural networks, gradient descent, multiplicative interactions, weight averaging, normalized gradients, and the like. These results are promising, but there is still much work to be done. In the remainder of this section, we motivate future work with concrete observations.

The search method was not the focus of this study but to reach our results, it helped to (1) add parallelism through migration, (2) use FEC, (3) increase diversity, and (4) apply hurdles, as we detailed in Section 3.2. The effects can be seen in Figure 8. Suppl. Section S9 shows that these improvements work across compute scales (today's high-compute regime is likely to be tomorrow's low-compute regime, so ideas that do not scale with compute will be shorter-lived). Preliminary implementations of crossover and geographic structure did not help in our experiments. The silver lining is that the AutoML-Zero search space provides ample room for algorithms to distinguish themselves (*e.g.* Section 4.1), allowing future work to attempt more sophisticated evolutionary approaches, reinforcement learning, Bayesian optimization, and other methods that have helped AutoML before.

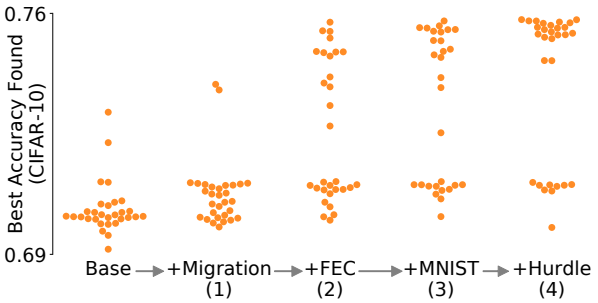


Figure 8: Search method ablation study. From left to right, each column adds an upgrade, as described in the main text.

Evaluating evolved algorithms on new tasks requires hyperparameter tuning, as is common for machine learning algorithms, but without inspection we may not know what each variable means (*e.g.* "Is `s7` the learning rate?"). Tuning all constants in the program was insufficient due to *hyperparameter coupling*, where an expression happens to produce a good value for a hyperparameter on a specific set of tasks but won't generalize. For example, evolution may choose `s7=v2·v2` because `v2·v2` coincides with a good value for the hyperparameter `s7`. We address this through manual decoupling (*e.g.* recognizing the problematic code line and instead setting `s7` to a constant that can be tuned later). This required time-consuming analysis that could be automated by future work. More details can be found in Suppl. Section S7.

Interpreting evolved algorithms also required effort due to the complexity of their raw code (Suppl. Section S8). The code was first automatically simplified by removing redundant instructions through static analysis. Then, to decide upon interesting code snippets, Section 4.3 focused on motifs that reappeared in independent search experiments. Such *convergent evolution* provided a hypothesis that a code section may be beneficial. To verify this hypothesis, we used ablations/*knock-outs* and *knock-ins*, the latter being the insertion of code sections into simpler algorithms to see if they are beneficial there too. This is analogous to homonymous molecular biology techniques used to study gene function. Further work may incorporate other techniques from the natural sciences or machine learning where interpretation of complex systems is key.

Search space enhancements have improved architecture search dramatically. In only two years, for example, comparable experiments went from requiring hundreds of GPUs (Zoph et al., 2018) to only one (Liu et al., 2019b). Similarly, enhancing the search space could bring significant improvements to AutoML-Zero. Also note that, despite our best effort to reduce human bias, there is still implicit bias in our current search space that limits the potential to discover certain types of algorithms. For instance, to keep our search space simple, we process one example at a time, so discovering techniques that work on batches of examples (like batch-norm) would require adding loops or higher-order tensors. As another case in point, in the current search space, a multi-layer neural network can only be found by discovering each layer independently; the addition of loops or function calls could make it easier to unlock such deeper structures.

Author Contributions

ER and QVL conceived the project; ER led the project; QVL provided advice; ER designed the search space, built the initial framework, and demonstrated plausibility; CL designed proxy tasks, built the evaluation pipeline, and analyzed the algorithms; DRS improved the search method and scaled up the infrastructure; ER, CL, and DRS ran the experiments; ER wrote the paper with contributions from CL; all authors edited the paper and prepared the figures; CL open-sourced the code.

Acknowledgements

We would like to thank Samy Bengio, Vincent Vanhoucke, Doug Eck, Charles Sutton, Yanping Huang, Jacques Pienaar, and Jeff Dean for helpful discussions, and especially Gabriel Bender, Hanxiao Liu, Rishabh Singh, Chiyuan Zhang, Hieu Pham, David Dohan and Alok Aggarwal for useful comments on the paper, as well as the larger Google Brain team.

References

Alba, E. and Tomassini, M. Parallelism and evolutionary algorithms. *IEEE transactions on evolutionary computation*, 2002.

Alet, F., Schneider, M. F., Lozano-Perez, T., and Kaelbling, L. P. Meta-learning curiosity algorithms. In *International Conference on Learning Representations*, 2019.

Andrychowicz, M., Denil, M., Gomez, S., Hoffman, M. W., Pfau, D., Schaul, T., and de Freitas, N. Learning to learn by gradient descent by gradient descent. In *NIPS*, 2016.

Angeline, P. J., Saunders, G. M., and Pollack, J. B. An evolutionary algorithm that constructs recurrent neural networks. *IEEE transactions on Neural Networks*, 1994.

Baker, B., Gupta, O., Naik, N., and Raskar, R. Designing neural network architectures using reinforcement learning. In *ICLR*, 2017.

Balog, M., Gaunt, A. L., Brockschmidt, M., Nowozin, S., and Tarlow, D. Deepcoder: Learning to write programs. *ICLR*, 2017.

Bello, I., Zoph, B., Vasudevan, V., and Le, Q. V. Neural optimizer search with reinforcement learning. *ICML*, 2017.

Bengio, S., Bengio, Y., and Cloutier, J. Use of genetic programming for the search of a new learning rule for neural networks. In *Evolutionary Computation*, 1994.

Bengio, Y. Practical recommendations for gradient-based training of deep architectures. In *Neural networks: Tricks of the trade*. Springer, 2012.

Bengio, Y., Léonard, N., and Courville, A. Estimating or propagating gradients through stochastic neurons for conditional computation. *arXiv preprint*, 2013.

Bergstra, J. and Bengio, Y. Random search for hyper-parameter optimization. *JMLR*, 2012.

Cai, H., Zhu, L., and Han, S. Proxylessnas: Direct neural architecture search on target task and hardware. *ICLR*, 2019.

Chalmers, D. J. The evolution of learning: An experiment in genetic connectionism. In *Connectionist Models*. Elsevier, 1991.

Chen, X., Liu, C., and Song, D. Towards synthesizing complex programs from input-output examples. *arXiv preprint arXiv:1706.01284*, 2017.

Chrabaszcz, P., Loshchilov, I., and Hutter, F. A downsampled variant of imagenet as an alternative to the cifar datasets. *arXiv preprint arXiv:1707.08819*, 2017.

Collins, M. Discriminative training methods for hidden markov models: Theory and experiments with perceptron algorithms. In *Proceedings of the ACL-02 conference on Empirical methods in natural language processing-Volume 10*, pp. 1–8. Association for Computational Linguistics, 2002.

Cubuk, E. D., Zoph, B., Mane, D., Vasudevan, V., and Le, Q. V. Autoaugment: Learning augmentation policies from data. *CVPR*, 2019a.

Cubuk, E. D., Zoph, B., Shlens, J., and Le, Q. V. Randaugment: Practical automated data augmentation with a reduced search space. *arXiv preprint arXiv:1909.13719*, 2019b.

Devlin, J., Uesato, J., Bhupatiraju, S., Singh, R., rahman Mohamed, A., and Kohli, P. Robustfill: Neural program learning under noisy i/o. In *ICML*, 2017.

Elsken, T., Metzen, J. H., and Hutter, F. Efficient multi-objective neural architecture search via lamarckian evolution. In *ICLR*, 2019a.

Elsken, T., Metzen, J. H., and Hutter, F. Neural architecture search. In *Automated Machine Learning*. Springer, 2019b.

Fahlman, S. E. and Lebiere, C. The cascade-correlation learning architecture. In *NIPS*, 1990.

Finn, C., Abbeel, P., and Levine, S. Model-agnostic meta-learning for fast adaptation of deep networks. In *ICML*, 2017.

Forsyth, R. et al. Beagle-a darwinian approach to pattern recognition. *Kybernetes*, 10(3):159–166, 1981.

Gaier, A. and Ha, D. Weight agnostic neural networks. In *NeurIPS*, 2019.

Ghiasi, G., Lin, T.-Y., and Le, Q. V. Nas-fpn: Learning scalable feature pyramid architecture for object detection. In *CVPR*, 2019.

Goldberg, D. E. and Deb, K. A comparative analysis of selection schemes used in genetic algorithms. *FOGA*, 1991.

Goodfellow, I., Bengio, Y., and Courville, A. *Deep learning*. MIT press, 2016.

Graves, A., Wayne, G., and Danihelka, I. Neural turing machines. *arXiv preprint arXiv:1410.5401*, 2014.

Gulwani, S., Polozov, O., Singh, R., et al. Program synthesis. *Foundations and Trends® in Programming Languages*, 2017.

Hazan, E., Levy, K., and Shalev-Shwartz, S. Beyond convexity: Stochastic quasi-convex optimization. In *Advances in Neural Information Processing Systems*, pp. 1594–1602, 2015.

He, K., Zhang, X., Ren, S., and Sun, J. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *ICCV*, 2015.

Hochreiter, S. and Schmidhuber, J. Long short-term memory. *Neural Computation*, 1997.

Holland, J. Adaptation in natural and artificial systems: an introductory analysis with application to biology. *Control and artificial intelligence*, 1975.

Hutter, F., Hoos, H. H., and Leyton-Brown, K. Sequential model-based optimization for general algorithm configuration. In *LION*, 2011.

Jaderberg, M., Dalibard, V., Osindero, S., Czarnecki, W. M., Donahue, J., Razavi, A., Vinyals, O., Green, T., Dunning, I., Simonyan, K., et al. Population based training of neural networks. *arXiv*, 2017.

Jayakumar, S. M., Menick, J., Czarnecki, W. M., Schwarz, J., Rae, J., Osindero, S., Teh, Y. W., Harley, T., and Pascanu, R. Multiplicative interactions and where to find them. In *ICLR*, 2020.

Kim, M. and Rigazio, L. Deep clustered convolutional kernels. *arXiv*, 2015.

Koza, J. R. and Koza, J. R. *Genetic programming: on the programming of computers by means of natural selection*. MIT press, 1992.

Krizhevsky, A. and Hinton, G. Learning multiple layers of features from tiny images. *Master's thesis, Dept. of Computer Science, U. of Toronto*, 2009.

Lake, B. M., Salakhutdinov, R., and Tenenbaum, J. B. Human-level concept learning through probabilistic program induction. *Science*, 350(6266):1332–1338, 2015.

LeCun, Y., Bengio, Y., et al. Convolutional networks for images, speech, and time series. *The handbook of brain theory and neural networks*, 1995.

LeCun, Y., Cortes, C., and Burges, C. J. The mnist database of handwritten digits, 1998.

Lenat, D. B. Eurisko: a program that learns new heuristics and domain concepts: the nature of heuristics iii: program design and results. *Artificial intelligence*, 21(1-2):61–98, 1983.

Levy, K. Y. The power of normalization: Faster evasion of saddle points. *arXiv preprint arXiv:1611.04831*, 2016.

Li, K. and Malik, J. Learning to optimize. *ICLR*, 2017.

Li, L. and Talwalkar, A. Random search and reproducibility for neural architecture search. *CoRR*, abs/1902.07638, 2019. URL http://arxiv.org/abs/1902.07638.

Li, L., Jamieson, K., DeSalvo, G., Rostamizadeh, A., and Talwalkar, A. Hyperband: A novel bandit-based approach to hyperparameter optimization. *JMLR*, 2018.

Liang, C., Berant, J., Le, Q. V., Forbus, K. D., and Lao, N. Neural symbolic machines: Learning semantic parsers on freebase with weak supervision. In *ACL*, 2016.

Liang, C., Norouzi, M., Berant, J., Le, Q. V., and Lao, N. Memory augmented policy optimization for program synthesis and semantic parsing. In *NeurIPS*, 2018.

Liu, C., Zoph, B., Shlens, J., Hua, W., Li, L.-J., Fei-Fei, L., Yuille, A., Huang, J., and Murphy, K. Progressive neural architecture search. *ECCV*, 2018.

Liu, C., Chen, L.-C., Schroff, F., Adam, H., Hua, W., Yuille, A. L., and Fei-Fei, L. Auto-deeplab: Hierarchical neural architecture search for semantic image segmentation. In *CVPR*, 2019a.

Liu, H., Simonyan, K., and Yang, Y. Darts: Differentiable architecture search. *ICLR*, 2019b.

Loshchilov, I. and Hutter, F. Cma-es for hyperparameter optimization of deep neural networks. *arXiv preprint arXiv:1604.07269*, 2016.

Mei, J., Li, Y., Lian, X., Jin, X., Yang, L., Yuille, A., and Yang, J. Atomnas: Fine-grained end-to-end neural architecture search. *ICLR*, 2020.

Mendoza, H., Klein, A., Feurer, M., Springenberg, J. T., and Hutter, F. Towards automatically-tuned neural networks. In *Workshop on Automatic Machine Learning*, 2016.

Metz, L., Maheswaranathan, N., Cheung, B., and Sohl-Dickstein, J. Meta-learning update rules for unsupervised representation learning. In *ICLR*, 2019.

Miikkulainen, R., Liang, J., Meyerson, E., Rawal, A., Fink, D., Francon, O., Raju, B., Shahrzad, H., Navruzyan, A., Duffy, N., et al. Evolving deep neural networks. In *Artificial Intelligence in the Age of Neural Networks and Brain Computing*. Elsevier, 2019.

Nair, V. and Hinton, G. E. Rectified linear units improve restricted boltzmann machines. In *ICML*, 2010.

Neelakantan, A., Le, Q. V., and Sutskever, I. Neural programmer: Inducing latent programs with gradient descent. *CoRR*, abs/1511.04834, 2015.

Negrinho, R., Gormley, M., Gordon, G. J., Patil, D., Le, N., and Ferreira, D. Towards modular and programmable architecture search. In *NeurIPS*, 2019.

Netzer, Y., Wang, T., Coates, A., Bissacco, A., Wu, B., and Ng, A. Y. Reading digits in natural images with unsupervised feature learning. 2011.

Noy, A., Nayman, N., Ridnik, T., Zamir, N., Doveh, S., Friedman, I., Giryes, R., and Zelnik-Manor, L. Asap: Architecture search, anneal and prune. *arXiv*, 2019.

Orchard, J. and Wang, L. The evolution of a generalized neural learning rule. In *IJCNN*, 2016.

Parisotto, E., rahman Mohamed, A., Singh, R., Li, L., Zhou, D., and Kohli, P. Neuro-symbolic program synthesis. *ArXiv*, abs/1611.01855, 2016.

Park, D. S., Chan, W., Zhang, Y., Chiu, C.-C., Zoph, B., Cubuk, E. D., and Le, Q. V. Specaugment: A simple data augmentation method for automatic speech recognition. *Proc. Interspeech*, 2019.

Pitrat, J. Implementation of a reflective system. *Future Generation Computer Systems*, 12(2-3):235–242, 1996.

Polozov, O. and Gulwani, S. Flashmeta: a framework for inductive program synthesis. In *OOPSLA 2015*, 2015.

Polyak, B. T. and Juditsky, A. B. Acceleration of stochastic approximation by averaging. *SIAM journal on control and optimization*, 30(4):838–855, 1992.

Ramachandran, P., Zoph, B., and Le, Q. Searching for activation functions. *ICLR Workshop*, 2017.

Ravi, S. and Larochelle, H. Optimization as a model for few-shot learning. *ICLR*, 2017.

Real, E., Moore, S., Selle, A., Saxena, S., Suematsu, Y. L., Le, Q., and Kurakin, A. Large-scale evolution of image classifiers. In *ICML*, 2017.

Real, E., Aggarwal, A., Huang, Y., and Le, Q. V. Regularized evolution for image classifier architecture search. *AAAI*, 2019.

Reed, S. E. and de Freitas, N. Neural programmer-interpreters. *CoRR*, abs/1511.06279, 2015.

Risi, S. and Stanley, K. O. Indirectly encoding neural plasticity as a pattern of local rules. In *International Conference on Simulation of Adaptive Behavior*, 2010.

Rumelhart, D. E., Hinton, G. E., and Williams, R. J. Learning representations by back-propagating errors. *Nature*, 1986.

Runarsson, T. P. and Jonsson, M. T. Evolution and design of distributed learning rules. In *ECNN*, 2000.

Schmidhuber, J. *Evolutionary principles in self-referential learning, or on learning how to learn: the meta-meta-... hook*. PhD thesis, Technische Universität München, 1987.

Schmidhuber, J. Optimal ordered problem solver. *Machine Learning*, 54(3):211–254, 2004.

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 2016.

Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., et al. Mastering the game of go without human knowledge. *Nature*, 2017.

Snoek, J., Larochelle, H., and Adams, R. P. Practical bayesian optimization of machine learning algorithms. In *NIPS*, 2012.

So, D. R., Liang, C., and Le, Q. V. The evolved transformer. In *ICML*, 2019.

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. Dropout: A simple way to prevent neural networks from overfitting. *JMLR*, 2014.

Stanley, K. O. and Miikkulainen, R. Evolving neural networks through augmenting topologies. *Evol. Comput.*, 2002.

Stanley, K. O., Clune, J., Lehman, J., and Miikkulainen, R. Designing neural networks through neuroevolution. *Nature Machine Intelligence*, 2019.

Suganuma, M., Shirakawa, S., and Nagao, T. A genetic programming approach to designing convolutional neural network architectures. In *GECCO*, 2017.

Sun, Y., Xue, B., Zhang, M., and Yen, G. G. Evolving deep convolutional neural networks for image classification. *IEEE Transactions on Evolutionary Computation*, 2019.

Tan, M., Chen, B., Pang, R., Vasudevan, V., Sandler, M., Howard, A., and Le, Q. V. Mnasnet: Platform-aware neural architecture search for mobile. In *CVPR*, 2019.

Valkov, L., Chaudhari, D., Srivastava, A., Sutton, C. A., and Chaudhuri, S. Houdini: Lifelong learning as program synthesis. In *NeurIPS*, 2018.

Vanschoren, J. Meta-learning. *Automated Machine Learning*, 2019.

Wang, R., Lehman, J., Clune, J., and Stanley, K. O. Paired open-ended trailblazer (poet): Endlessly generating increasingly complex and diverse learning environments and their solutions. *GECCO*, 2019.

Wichrowska, O., Maheswaranathan, N., Hoffman, M. W., Colmenarejo, S. G., Denil, M., de Freitas, N., and Sohl-Dickstein, J. Learned optimizers that scale and generalize. *ICML*, 2017.

Wilson, D. G., Cussat-Blanc, S., Luga, H., and Miller, J. F. Evolving simple programs for playing atari games. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pp. 229–236, 2018.

Wu, Y., Schuster, M., Chen, Z., Le, Q. V., Norouzi, M., et al. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv*, 2016.

Xiao, H., Rasul, K., and Vollgraf, R. Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms. *arXiv preprint arXiv:1708.07747*, 2017.

Xie, L. and Yuille, A. Genetic CNN. In *ICCV*, 2017.

Xie, S., Kirillov, A., Girshick, R., and He, K. Exploring randomly wired neural networks for image recognition. In *Proceedings of the IEEE International Conference on Computer Vision*, pp. 1284–1293, 2019.

Yang, A., Esperança, P. M., and Carlucci, F. M. Nas evaluation is frustratingly hard. *ICLR*, 2020.

Yao, Q., Wang, M., Chen, Y., Dai, W., Yi-Qi, H., Yu-Feng, L., Wei-Wei, T., Qiang, Y., and Yang, Y. Taking human out of learning applications: A survey on automated machine learning. *arXiv*, 2018.

Yao, X. Evolving artificial neural networks. *IEEE*, 1999.

Ying, C., Klein, A., Real, E., Christiansen, E., Murphy, K., and Hutter, F. Nas-bench-101: Towards reproducible neural architecture search. *ICML*, 2019.

Zela, A., Klein, A., Falkner, S., and Hutter, F. Towards automated deep learning: Efficient joint neural architecture and hyperparameter search. *ICML AutoML Workshop*, 2018.

Zhong, Z., Yan, J., Wu, W., Shao, J., and Liu, C.-L. Practical block-wise neural network architecture generation. In *CVPR*, 2018.

Zoph, B. and Le, Q. V. Neural architecture search with reinforcement learning. In *ICLR*, 2016.

Zoph, B., Vasudevan, V., Shlens, J., and Le, Q. V. Learning transferable architectures for scalable image recognition. In *CVPR*, 2018.