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.* 30, 75, 90]. 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 [72, 39, 31, 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. 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 [102, 69, 84]. Other AutoML studies similarly have found ways to constrain their search spaces to isolated algorithmic aspects, such as the learning rule used during backpropagation [2, 67], the gating structure of an LSTM [6, 34], or the data augmentation [17, 62]; 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: you cannot discover what you cannot search for [21]. Indeed, dominant aspects of performance are often left out [94]. Second, constrained search spaces need to be carefully composed [102, 78, 57], thus creating a new burden on researchers and curtailing the purported objective of saving human 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.* 76]. 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 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

[*] Equal contribution [1] Google Brain/Google Research, Mountain View, CA, USA. Correspondence to: Esteban Real <ereal@google.com>. Copyright 2020 by the author(s).

that advanced techniques are often only marginally superior to simple random search (RS) [43, 21, 57]. 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* that predict and learn from one example at a time. 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 datasets as proxy tasks and migration techniques to build highly-optimized open-source infrastructure capable of searching through 10,000 models/second/cpu.[1]. 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.

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 (Supplementary Section S9).

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

Because our approach simultaneously searches all the aspects of an ML algorithm, it relates to previous work that targets each aspect individually. As there are many such aspects (*e.g.* architecture, hyperparameters, learning rule), previous work is extensive and impossible to exhaustively list here. Many examples belong within the field of *AutoML*. A frequently targeted aspect of the ML algorithm is the structure of the model; this is known as *architecture search*. It has a long history [22, 3, 96, 80, 11, 52, 4, 101, 68, 92, 82, 47, and many others] and continues today [49, 20, 12, 48, 24, 83, 93, and many others]. Reviews provide more thorough background [21, 81, 95]. Recent work has obtained accurate models by constraining the space to only look for the structure of a block that is then stacked to form a neural network. The stacking is fixed and the block is free to combine standard neural network layers into patterns that optimize the accuracy of the model [102, 100]. Mei et al. [51] highlight the importance of finer-grained search spaces and take a step in that direction by splitting convolutions into channels that can be handled separately. Other specific architecture aspects have also been targeted, such as the hyperparameters [77, 50, 32, 44], activation functions [66], a specific layer [35], the full forward pass [23], the data augmentation [17, 62, 18], *etc*. Beyond these narrowly targeted search spaces, *more* inclusive spaces are already demonstrating promise. For example, a few studies have combined two seemingly disparate algorithmic aspects into a single search space: the inner modules and the outer structure [54], the architecture and the hyperparameters [99], the layers and the weight pruning [59], and so on. We extend this to all aspects of the algorithm, including the weight optimization.

Searching for a better optimizer for the neural network's weights has been a research topic for decades. Chalmers [13] formalizes the update rule for the weights as $w_{i,j} \leftarrow w_{i,j} + F(x_1, x_2, ...)$, where x_i are local signals and F combines them linearly. The coefficients of the linear combination constitute the search space and are encoded as a bit string that is searched with a genetic algorithm. This is an example of a *numerically learned* update rule: the final result is a set of coefficients that work very well but may not be interpretable. Numerically learned optimizers have improved since then. Studies found that Chalmers' F formula above can be replaced with more advanced struc-

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

tures, such as a second neural network [73, 60], an LSTM [67], a hierarchical RNN [88], or even a different LSTM for each weight [53]. Numerically or otherwise, some studies center on the method by which the optimizer is learned; it can vary widely from the use of gradient descent [2] to reinforcement learning [42] to evolutionary search with sophisticated developmental encodings [71]. All these methods are sometimes collectively labeled as *meta-learning* [86] or described as "learning the learning algorithm", as the optimizer is indeed an algorithm. However, in this work, we understand *algorithm* more broadly and it will include the neural network itself. Additionally, our algorithm is not learned numerically, but discovered *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. [8], who represent F as a tree: the leaves are the possible inputs to the optimizer (*i.e.* the x_i above) and the nodes are one of $\{+, -, \times, \div\}$. F is then evolved, making this an example of *genetic programming* [36]. Our search method is similar to genetic programming but we choose to represent the program as a sequence of instructions—like a programmer would type it—rather than a tree. Another similarity with Bengio et al. [8] is that they also use simple mathematical operations as building blocks. We use many more, however, including vector and matrix instructions that take advantage of dense hardware computations. More recently, Bello et al. [7] revisited symbolically learned optimizers to apply them to a modern neural network. Their goal was to maximize the final accuracy of their models and so they restrict the search space by allowing 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. Both Bengio et al. [8] and Bello et al. [7] assume the existence of a neural network with a forward pass that computes the activations and a backward pass that provides the weight gradients. Thus, the search process can just focus on discovering how to use these activations and gradients to adjust the network's weights. In contrast, we do not assume the existence of a network. It must therefore be discovered, as must the weights, gradients and update code.

More loosely, our work also relates to program synthesis, where the application of ML has grown recently [28]. The synthesized programs—or algorithms—solve problems like sorting, addition, counting [74, 27, 70, 85], string manipulations [64, 61, 19], character recognition [38], competition-style programming [5], structured data QA [56, 45, 46], program parsing [14], and game playing [89], to name a few. Unlike these, we focus on synthesizing code that solves the problem of *doing* ML.

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 search could be random, evolutionary, *etc*. It measures the quality of the algorithms on a subset $\mathcal{T}_{search} \subset \mathcal{T}$, and each experiment produces a high-quality candidate algorithm. Once the search is done, we select the best of the candidates by measuring their performance 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 [37]. To lower compute cost and achieve higher throughput, we use random projections to reduce the dimensionality of the features to create small proxy tasks for $\mathcal{T}_{search}$ and $\mathcal{T}_{select}$. 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 features in the original dimensionality (3072), holding out the CIFAR-10 test set for the final evaluation. To make sure the improvement over baselines is not specific to CIFAR-10, we further show the gain generalizes to other datasets, SVHN [58], ImageNet [15], and Fashion MNIST [91]. 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 in 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). We represent a program as a sequence 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 used 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. Supplementary Section S1 contains the full list of 65 ops.

Inspired by recent 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). These are interpreted by the evaluation process in Fig 1. There, the two for-loops implement the *training* and *validation phases*, processing 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 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 evaluation then takes the median across D tasks. This median accuracy is used as a measure of the algorithm's quality by the search method.

```
# (Setup, Predict, Learn) is the input ML algorithm.
# Dtrain / Dvalid is the 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 execute 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.

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 [69, 97, 78]. 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*, [25]). 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

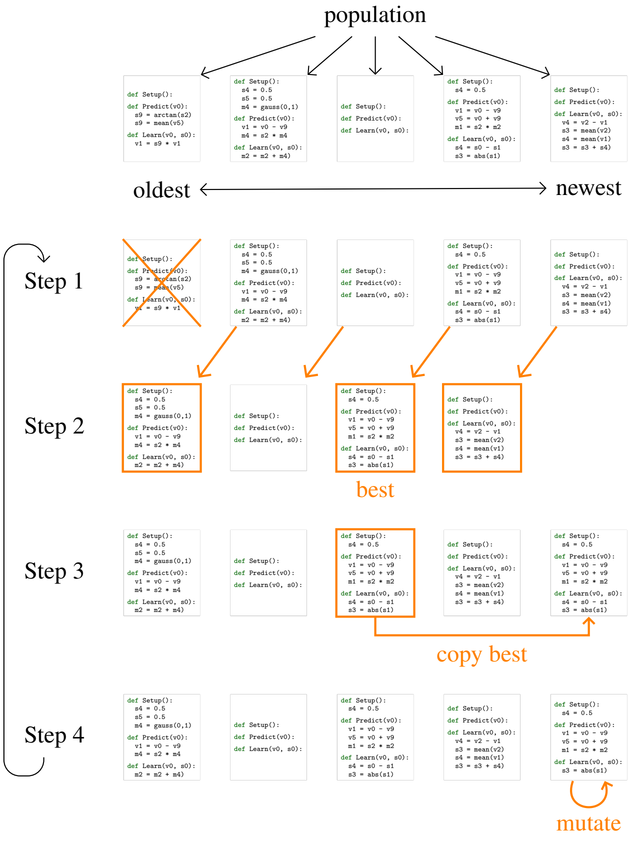
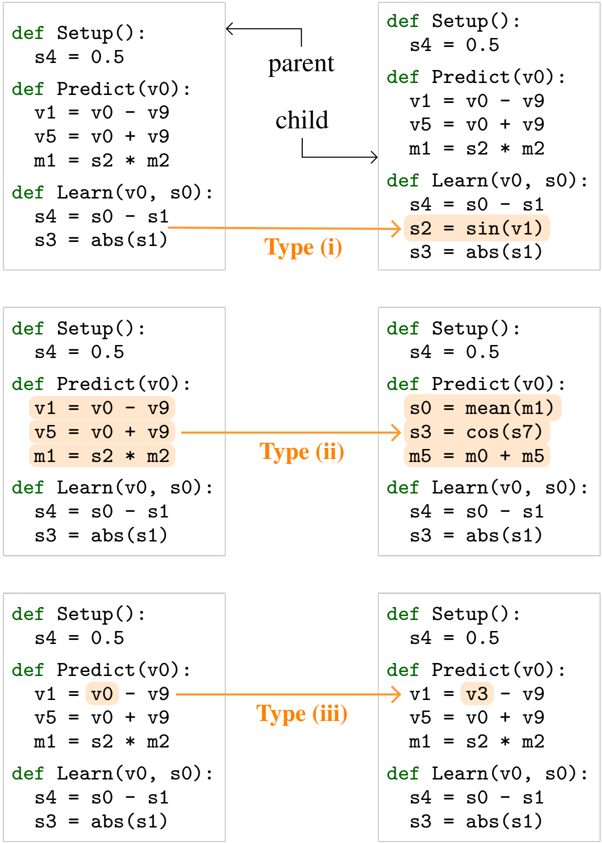


Figure 2: One cycle of the evolutionary method [25, 69]. A population of P algorithms (here, P=5; laid out from left to right in the order they were discovered) undergoes many cycles like this one. First, we remove the oldest algorithm (step 1). Then, we choose a random subset of size T (here, T=3) and select the best of them (step 2). The best is copied (step 3) and mutated (step 4).



```
def Setup():
  s4 = 0.5
def Predict(v0):
  v1 = v0 - v9
  v5 = v0 + v9
  m1 = s2 * m2
def Learn(v0, s0):
  s4 = s0 - s1
  s3 = abs(s1)
```

```
def Setup():
  s4 = 0.5
def Predict(v0):
  v1 = v0 - v9
  v5 = v0 + v9
  m1 = s2 * m2
def Learn(v0, s0):
  s4 = s0 - s1
  s2 = sin(v1)
  s3 = abs(s1)
```

```
def Setup():
  s4 = 0.5
def Predict(v0):
  v1 = v0 - v9
  v5 = v0 + v9
  m1 = s2 * m2
def Learn(v0, s0):
  s4 = s0 - s1
  s3 = abs(s1)
```

```
def Setup():
  s4 = 0.5
def Predict(v0):
  s0 = mean(m1)
  s3 = cos(s7)
  m5 = m0 + m5
def Learn(v0, s0):
  s4 = s0 - s1
  s3 = abs(s1)
```

```
def Setup():
  s4 = 0.5
def Predict(v0):
  v1 = v0 - v9
  v5 = v0 + v9
  m1 = s2 * m2
def Learn(v0, s0):
  s4 = s0 - s1
  s3 = abs(s1)
```

```
def Setup():
  s4 = 0.5
def Predict(v0):
  v1 = v3 - v9
  v5 = v0 + v9
  m1 = s2 * m2
def Learn(v0, s0):
  s4 = s0 - s1
  s3 = abs(s1)
```

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.

constant”). These are illustrated in Figure 3.

In order to reach a throughput of 2k–10k algorithms/second/cpu, 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 and avoid unnecessary evaluations. (2) We add hurdles [78] 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 [1]. 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. 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.* [87]). Section 5 and Supplementary Section S8 contain ablation studies showing that all these upgrades 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. *Num. worker processes* W=1k *or* 10k, *unless otherwise stated. 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 S4. Suppl. Section S2 describes further 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?”

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.); For fairness, RS expts. match the*

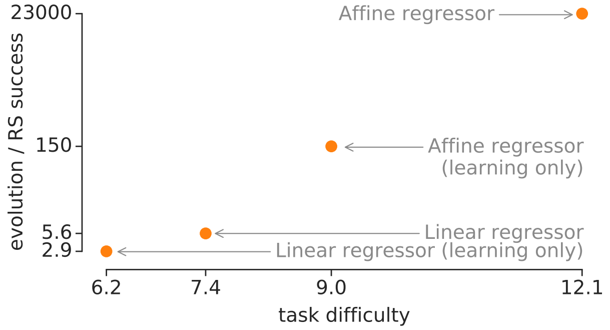


Figure 4: Relative success rate of evolution and random search (RS). Each point represents a different task type and the x-axis measures its difficulty (defined in main text). As the task type becomes more difficult, evolution vastly outperforms RS, illustrating the complexity of AutoML-Zero when compared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

*combined resources of evolution. We repeat expts. until statistical significance is achieved. Full configs. in Suppl. Section S4. 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 that perform equal to or better than a hand-designed reference model (*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 the log of 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 in every $10^{7.4}$, so we define the difficulty of the linear regression task type to be 7.4. We then run the evolution experiments. As in RS, we measure the ratio of acceptable algorithms to the total number of algorithms evaluated, to measure 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 acceptable algorithms that are all copies of each other. 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 other AutoML studies, as highlighted in Section 1, where the solutions are dense enough that RS can be competitive.

Figure 4 summarizes the result of this analysis for 4 task types: the discovery of a full-algorithm/only-the-learning 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: $L(x_i) = u \cdot ReLU(Wx_i)$, where W 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 S4. Note that the restrictions above apply *only* to this section (4.1).*____

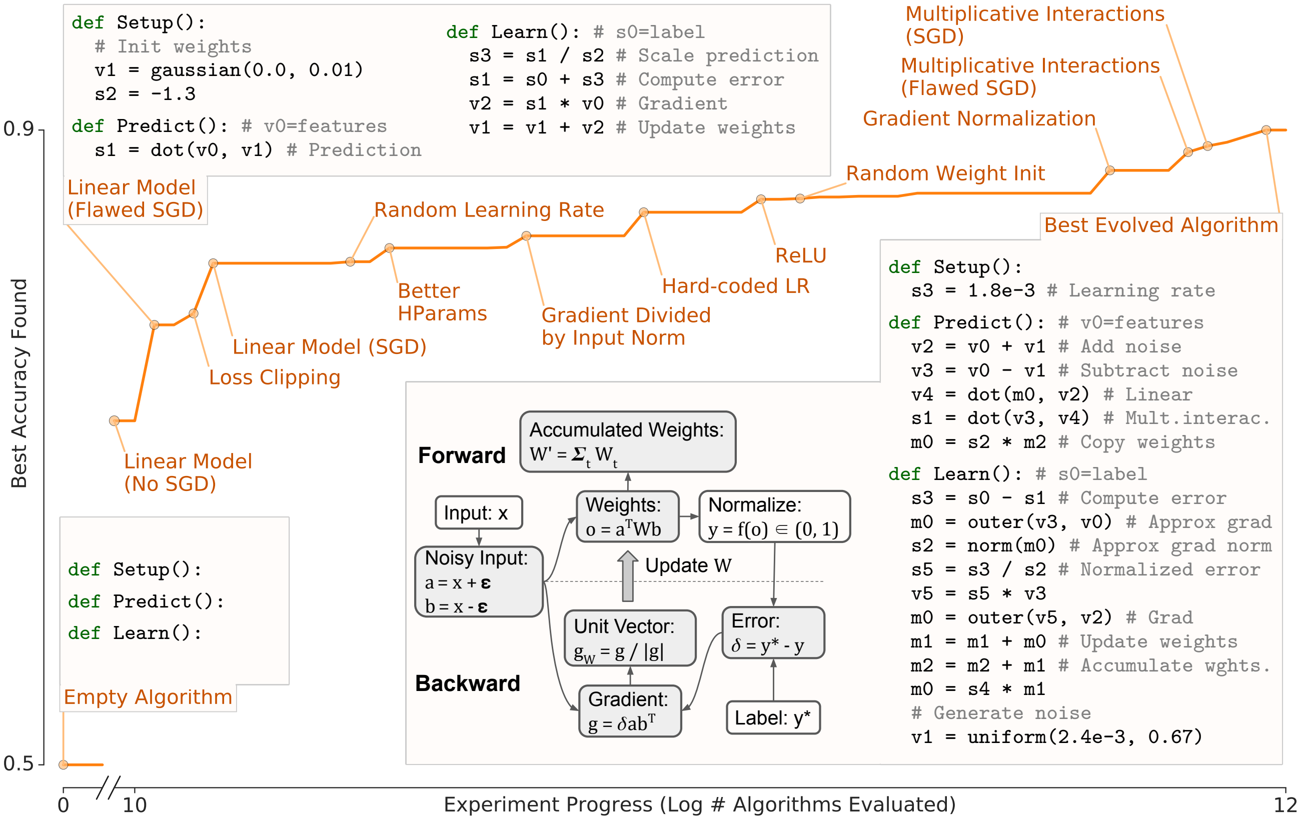
When the search method uses only 1 task in $\mathcal{T}_{search}$ (*i.e.* D=1), the algorithm evolves the exact prediction function used by the teacher and hard-codes its weights. The results become more surprising as we increase the number of tasks in $\mathcal{T}_{search}$ (*e.g.* to D=100), as now 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.

```
# sX/vX/mX = scalar/vector/matrix at address X.
# The C_ (eg C1) are constants tuned by search.
# "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.

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



```
def Setup():
def Predict():
def Learn():
```

```
def Setup():
  # Init weights
  v1 = gaussian(0.0, 0.01)
  s2 = -1.3
def Predict(): # v0=features
  s1 = dot(v0, v1) # Prediction

def Learn(): # s0=label
  s3 = s1 / s2 # Scale prediction
  s1 = s0 + s3 # Compute error
  v2 = s1 * v0 # Gradient
  v1 = v1 + v2 # Update weights
```

```
def Setup():
  s3 = 1.8e-3 # Learning rate
def Predict(): # v0=features
  v2 = v0 + v1 # Add noise
  v3 = v0 - v1 # Subtract noise
  v4 = dot(m0, v2) # Linear
  s1 = dot(v3, v4) # Mult.interac.
  m0 = s2 * m2 # Copy weights
def Learn(): # s0=label
  s3 = s0 - s1 # Compute error
  m0 = outer(v3, v0) # Approx grad
  s2 = norm(m0) # Approx grad norm
  s5 = s3 / s2 # Normalized error
  v5 = s5 * v3
  m0 = outer(v5, v2) # Grad
  m1 = m1 + m0 # Update weights
  m2 = m2 + m1 # Accumulate wghts.
  m0 = s4 * m1
  # Generate noise
  v1 = uniform(2.4e-3, 0.67)
```

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 through 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.

other than neural networks. For more realistic datasets, we use binary classification tasks extracted from CIFAR-10 and MNIST.

____ *Experiment Details: We extracted tasks from the CIFAR-10 and MNIST training sets; each of the datasets were 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* $\mathcal{T}_{select}$*, ı.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. 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 S4.*_

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 original CIFAR-10 data.

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, $82.22 \pm 0.17\%$). The gain also generalizes to binary classification tasks extracted from other datasets: SVHN [58] (88.12% for the best evolved algorithm *vs.* 59.58% for the linear baseline *vs.* 85.14% for the nonlinear baseline), downsampled ImageNet [15] (80.78% *vs.* 76.44% *vs.* 78.44%), Fashion MNIST [91] (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 Supplementary Section S5). The algorithm has the following notable features, whose usefulness we verified through ablations (more details in Supplementary Section S7): (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 an uniform distribution. (2) multiplicative interactions [33] 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. The normalized gradient is a commonly used heuristic in non-convex optimization [29, 41], which could help with the vanishing and exploding gradient problem [98, 63]. (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.*:

$$\mathbf{W'} = \sum_t \mathbf{W_t}$$

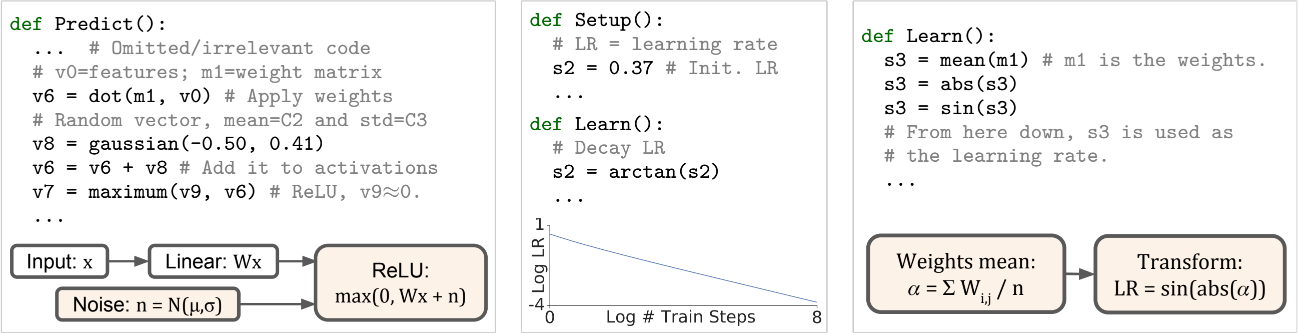
This is reminiscent of the *averaged perceptron* [16] and neural network *weight averaging* during training [65, 26]. Unlike these studies, the evolved algorithm *accumulates* instead of averaging, but this difference has no effect when measuring the accuracy of classification tasks (it does not change 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` execute in alternation. However, during validation, `Learn` is never called and `Predict` is executed repeatedly, causing $\mathbf{W}$ to remain as $\mathbf{W'}$.

In conclusion, the aggregate metrics and case study presented show that our framework can discover algorithms that reach the level of human designs, 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 challenge. 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 fol-*



```
def Predict():
  ...  # Omitted/irrelevant code
  # v0=features; m1=weight matrix
  v6 = dot(m1, v0) # Apply weights
  # Random vector, mean=C2 and std=C3
  v8 = gaussian(-0.50, 0.41)
  v6 = v6 + v8 # Add it to activations
  v7 = maximum(v9, v6) # ReLU, v9≈0.
  ...
```

```
def Setup():
  # LR = learning rate
  s2 = 0.37 # Init. LR
  ...
def Learn():
  # Decay LR
  s2 = arctan(s2)
  ...
```

```
def Learn():
  s3 = mean(m1) # m1 is the weights.
  s3 = abs(s3)
  s3 = sin(s3)
  # From here down, s3 is used as
  # the learning rate.
  ...
```

(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 S5.

lowing exceptions: F=16*;* $10 \leq D \leq 100$*; critical alterations to the data are explained in each task type below. Full configs. in Suppl. Section S4.*

Few training examples. We use only 80 of the examples and repeat them for 100 epochs. Under these conditions, algorithms evolved an adaptation that augments the data through the injection of noise (Figure 7a). This is referred to in the literature as a *noisy ReLU* [55, 10] and is reminiscent of Dropout [79]. 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 ($p < 0.0005$, N=60).

Fast training. Training on 800 examples/10 epochs led to the repeated emergence of learning-rate decay, a well-known strategy for the timely training of an ML model [9]. An example can be seen in Figure 7b. As a control, we increased 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).

Multiple classes. In this case, we use all 10 classes of the CIFAR-10 dataset. Evolved algorithms sometimes use the transformed mean of the weight matrix as the learning rate (Figure 7c). We do not know why this mechanism would benefit the multi-class task more than the binary-class task, but the preference is statistically significant (expt: 24/30, control: 0/30, *i.e.* vanishingly small p-value).

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

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 linear regressors, neural networks, gradient descent, multiplicative interactions, weight averaging, normalized gradients, etc. 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. Supplementary Section S8 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). This affords an opportunity for future work to improve upon our results with more sophisticated evolutionary approaches, reinforcement learning, Bayesian optimization, and other methods that have helped AutoML before.

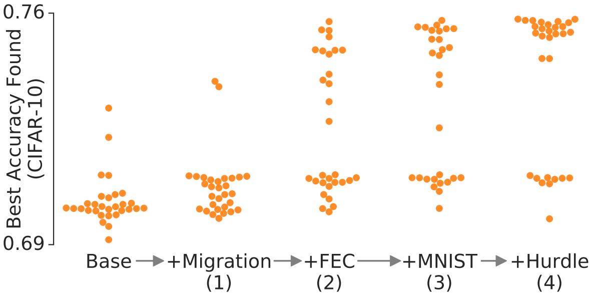


Figure 8: Search method ablation study. From left to right, each column adds an upgrade, as described in the main text.

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 Supplementary Section S6.

Interpreting evolved algorithms also required effort due to the complexity of their raw code (Supplementary Section S7). 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 [102] to only one [49]. Similarly, carefully designing the search space could bring significant improvements to AutoML-Zero. For simplicity, our framework processes one example at a time, but the addition of loops or higher-order tensors would allow representing operations that work on batches of examples—like batch-norm. Function calls may help too. At the moment, a deep neural network can only appear by discovering layer after layer, even if all the layers are identical. The addition of function calls may unlock 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, and Hieu Pham for useful comments on the paper, as well as the larger Google Brain team.

References

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

[2] 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.

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

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

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

[6] Bayer, J., Wierstra, D., Togelius, J., and Schmidhuber, J. Evolving memory cell structures for sequence learning. In *ICANN*, 2009.

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

[8] 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.

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

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

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

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

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

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

[15] 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.

[16] 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.

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

[18] 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*, 2019.

[19] 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.

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

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

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

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

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

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

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

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

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

[29] 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.

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

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

[32] 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.

[33] 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.

[34] Jozefowicz, R., Zaremba, W., and Sutskever, I. An empirical exploration of recurrent network architectures. In *ICML*, 2015.

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

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

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

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

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

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

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

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

[43] 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`.

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

[45] 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.

[46] 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.

[47] 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.

[48] 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*, 2019.

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

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

[51] 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.

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

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

[54] 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.

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

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

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

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

[59] 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.

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

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

[62] 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.

[63] Pascanu, R., Mikolov, T., and Bengio, Y. On the difficulty of training recurrent neural networks. In *International conference on machine learning*, pp. 1310–1318, 2013.

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

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

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

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

[68] 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.

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

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

[71] 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.

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

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

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

[75] 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.

[76] 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.

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

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

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

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

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

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

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

[84] 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.

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

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

[87] 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.

[88] 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.

[89] 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.

[90] 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.

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

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

[93] 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.

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

[95] 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.

[96] Yao, X. Evolving artificial neural networks. *IEEE*, 1999.

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

[98] Yu, A. W., Huang, L., Lin, Q., Salakhutdinov, R., and Carbonell, J. Block-normalized gradient method: An empirical study for training deep neural network. *arXiv preprint arXiv:1707.04822*, 2017.

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

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

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

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

AutoML-Zero: Evolving Machine Learning Algorithms From Scratch

Supplementary Material

S1. Search Space Additional Details

Supplementary Table S1 describes all the ops in our search space. They are ordered to reflect how we chose them: we imagined a typical school curriculum up to—but not including—calculus (see braces to the right of the table). In particular, there are no derivatives so any gradient computation used for training must be evolved.

S2. Search Method Additional Details

The mutations that produce the child from the parent must be tailored to the search space. We use a uniformly random choice among the following three transformations: (i) add or remove an instruction; instructions are added at a random position and have a random op and random arguments; to prevent programs from growing unnecessarily,

Table S1: Ops vocabulary. s, $\vec{v}$ and M denote a scalar, vector, and matrix, resp. Early-alphabet letters (a, b, *etc.*) denote memory addresses. Mid-alphabet letters (*e.g.* i, j, *etc.*) denote vector/matrix indexes ("Index" column). Greek letters denote constants ("Consts." column). $\mathcal{U}(\alpha, \beta)$ denotes a sample from a uniform distribution in $[\alpha, \beta]$. $\mathcal{N}(\mu, \sigma)$ is analogous for a normal distribution with mean μ and standard deviation σ. $\mathbb{1}_X$ is the indicator function for set X. Example: "$M_a^{(i,j)} = \mathcal{U}(\alpha, \beta)$" describes the operation "assign to the i,j-th entry of the matrix at address a a value sampled from a uniform random distribution in $[\alpha, \beta]$".

Op	Code	Input Args		Output Args		Description	
ID	Example	Addresses / types	Consts.	Address / type	Index	(see caption)	
OP0	`no_op`	–	–	–	–	–	
OP1	`s2=s3+s0`	a,b / scalars	–	c / scalar	–	$s_c = s_a + s_b$	Arithmetic
OP2	`s4=s0-s1`	a,b / scalars	–	c / scalar	–	$s_c = s_a - s_b$	
OP3	`s8=s5*s5`	a,b / scalars	–	c / scalar	–	$s_c = s_a\, s_b$	
OP4	`s7=s5/s2`	a,b / scalars	–	c / scalar	–	$s_c = s_a / s_b$	
OP5	`s8=abs(s0)`	a / scalar	–	b / scalar	–	$s_b = \|s_a\|$	
OP6	`s4=1/s8`	a / scalar	–	b / scalar	–	$s_b = 1/s_a$	
OP7	`s5=sin(s4)`	a / scalar	–	b / scalar	–	$s_b = \sin(s_a)$	Trigonometry
OP8	`s1=cos(s4)`	a / scalar	–	b / scalar	–	$s_b = \cos(s_a)$	
OP9	`s3=tan(s3)`	a / scalar	–	b / scalar	–	$s_b = \tan(s_a)$	
OP10	`s0=arcsin(s4)`	a / scalar	–	b / scalar	–	$s_b = \arcsin(s_a)$	
OP11	`s2=arccos(s0)`	a / scalar	–	b / scalar	–	$s_b = \arccos(s_a)$	
OP12	`s4=arctan(s0)`	a / scalar	–	b / scalar	–	$s_b = \arctan(s_a)$	
OP13	`s1=exp(s2)`	a / scalar	–	b / scalar	–	$s_b = e^{s_a}$	Pre-Calculus
OP14	`s0=log(s3)`	a / scalar	–	b / scalar	–	$s_b = \log s_a$	
OP15	`s3=heaviside(s0)`	a / scalar	–	b / scalar	–	$s_b = \mathbb{1}_{\mathbb{R}^+}(s_a)$	
OP16	`v2=heaviside(v2)`	a / vector	–	b / vector	–	$\vec{v}_b^{(i)} = \mathbb{1}_{\mathbb{R}^+}(\vec{v}_a^{(i)})\ \forall i$	
OP17	`m7=heaviside(m3)`	a / matrix	–	b / matrix	–	$M_b^{(i,j)} = \mathbb{1}_{\mathbb{R}^+}(M_a^{(i,j)})\ \forall i, j$	
OP18	`v1=s7*v1`	a,b / sc,vec	–	c / vector	–	$\vec{v}_c = s_a\, \vec{v}_b$	Linear Algebra
OP19	`v1=bcast(s3)`	a / scalar	–	b / vector	–	$\vec{v}_b^{(i)} = s_a \quad \forall i$	
OP20	`v5=1/v7`	a / vector	–	b / vector	–	$\vec{v}_b^{(i)} = 1/\vec{v}_a^{(i)} \quad \forall i$	
OP21	`s0=norm(v3)`	a / scalar	–	b / vector	–	$s_b = \|\vec{v}_a\|$	
OP22	`v3=abs(v3)`	a / vector	–	b / vector	–	$\vec{v}_b^{(i)} = \|\vec{v}_a^{(i)}\| \quad \forall i$	

[Table continues on the next page.]

Table S1: Ops vocabulary (continued)

Op ID	Code Example	Input Args		Output Args		Description	
		Addresses / types	Consts	Address / type	Index	(see caption)	
OP23	`v5=v0+v9`	a,b / vectors	–	c / vector	–	$\vec{v}_c = \vec{v}_a + \vec{v}_b$	Linear Algebra
OP24	`v1=v0-v9`	a,b / vectors	–	c / vector	–	$\vec{v}_c = \vec{v}_a - \vec{v}_b$	
OP25	`v8=v1*v9`	a,b / vectors	–	c / vector	–	$\vec{v}_c^{(i)} = \vec{v}_a^{(i)}\,\vec{v}_b^{(i)}\ \ \forall i$	
OP26	`v9=v8/v2`	a,b / vectors	–	c / vector	–	$\vec{v}_c^{(i)} = \vec{v}_a^{(i)} / \vec{v}_b^{(i)}\ \ \forall i$	
OP27	`s6=dot(v1,v5)`	a,b / vectors	–	c / scalar	–	$s_c = \vec{v}_a^T\,\vec{v}_b$	
OP28	`m1=outer(v6,v5)`	a,b / vectors	–	c / matrix	–	$M_c = \vec{v}_a\,\vec{v}_b^T$	
OP29	`m1=s4*m2`	a,b / sc/mat	–	c / matrix	–	$M_c = s_a\,M_b$	
OP30	`m3=1/m0`	a / matrix	–	b / matrix	–	$M_b^{(i,j)} = 1/M_a^{(i,j)}\ \ \forall i,j$	
OP31	`v6=dot(m1,v0)`	a,b / mat/vec	–	c / vector	–	$\vec{v}_c = M_a\,\vec{v}_b$	
OP32	`m2=bcast(v0,axis=0)`	a / vector	–	b / matrix	–	$M_b^{(i,j)} = \vec{v}_a^{(i)}\ \ \forall i,j$	
OP33	`m2=bcast(v0,axis=1)`	a / vector	–	b / matrix	–	$M_b^{(j,i)} = \vec{v}_a^{(i)}\ \ \forall i,j$	
OP34	`s2=norm(m1)`	a / matrix	–	b / scalar	–	$s_b = \|\|M_a\|\|$	
OP35	`v4=norm(m7,axis=0)`	a / matrix	–	b / vector	–	$\vec{v}_b^{(i)} = \|M_a^{(i,\cdot)}\|\ \ \forall i$	
OP36	`v4=norm(m7,axis=1)`	a / matrix	–	b / vector	–	$\vec{v}_b^{(j)} = \|M_a^{(\cdot,j)}\|\ \ \forall j$	
OP37	`m9=transpose(m3)`	a / matrix	–	b / matrix	–	$M_b = \|M_a^T\|$	
OP38	`m1=abs(m8)`	a / matrix	–	b / matrix	–	$M_b^{(i,j)} = \|M_a^{(i,j)}\|\ \ \forall i,j$	
OP39	`m2=m2+m0`	a,b / matrixes	–	c / matrix	–	$M_c = M_a + M_b$	
OP40	`m2=m3+m1`	a,b / matrixes	–	c / matrix	–	$M_c = M_a - M_b$	
OP41	`m3=m2*m3`	a,b / matrixes	–	c / matrix	–	$M_c^{(i,j)} = M_a^{(i,j)}\,M_b^{(i,j)}\ \ \forall i,j$	
OP42	`m4=m2/m4`	a,b / matrixes	–	c / matrix	–	$M_c^{(i,j)} = M_a^{(i,j)} / M_b^{(i,j)}\ \ \forall i,j$	
OP43	`m5=matmul(m5,m7)`	a,b / matrixes	–	c / matrix	–	$M_c = M_a\,M_b$	
OP44	`s1=minimum(s2,s3)`	a,b / scalars	–	c / scalar	–	$s_c = \min(s_a, s_b)$	Probability and Statistics
OP45	`v4=minimum(v3,v9)`	a,b / vectors	–	c / vector	–	$\vec{v}_c^{(i)} = \min(\vec{v}_a^{(i)}, \vec{v}_b^{(i)})\ \ \forall i$	
OP46	`m2=minimum(m2,m1)`	a,b / matrixes	–	c / matrix	–	$M_c^{(i,j)} = \min(M_a^{(i,j)}, M_b^{(i,j)})\ \ \forall i,j$	
OP47	`s8=maximum(s3,s0)`	a,b / scalars	–	c / scalar	–	$s_c = \max(s_a, s_b)$	
OP48	`v7=maximum(v3,v6)`	a,b / vectors	–	c / vector	–	$\vec{v}_c^{(i)} = \max(\vec{v}_a^{(i)}, \vec{v}_b^{(i)})\ \ \forall i$	
OP49	`m7=maximum(m1,m0)`	a,b / matrixes	–	c / matrix	–	$M_c^{(i,j)} = \max(M_a^{(i,j)}, M_b^{(i,j)})\ \ \forall i,j$	
OP50	`s2=mean(v2)`	a / vector	–	b / scalar	–	$s_b = \mathrm{mean}(\vec{v}_a)$	
OP51	`s2=mean(m8)`	a / matrix	–	b / scalar	–	$s_b = \mathrm{mean}(M_a)$	
OP52	`v1=mean(m2,axis=0)`	a / matrix	–	b / vector	–	$\vec{v}_b^{(i)} = \mathrm{mean}(M_a^{(i,\cdot)})\ \ \forall i$	
OP53	`v3=std(m2,axis=0)`	a / matrix	–	b / vector	–	$\vec{v}_b^{(i)} = \mathrm{stdev}(M_a^{(i,\cdot)})\ \ \forall i$	
OP54	`s3=std(v3)`	a / vector	–	b / scalar	–	$s_b = \mathrm{stdev}(\vec{v}_a)$	
OP55	`s4=std(m0)`	a / matrix	–	b / scalar	–	$s_b = \mathrm{stdev}(M_a)$	
OP56	`s2=0.1`	–	γ	a / scalar	–	$s_a = \gamma$	
OP57	`v3[5]=-2.4`	–	γ	a / vector	i	$\vec{v}_a^{(i)} = \gamma$	
OP58	`m2[5,1]=-0.03`	–	γ	a / matrix	i, j	$M_a^{(i,j)} = \gamma$	
OP59	`s4=uniform(-1,1)`	–	α, β	a / scalar	–	$s_a = \mathcal{U}(\alpha, \beta)$	
OP60	`v1=uniform(0.4,0.8)`	–	α, β	a / vector	–	$\vec{v}_a^{(i)} = \mathcal{U}(\alpha, \beta)\ \ \forall i$	

....[Table continues on the next page.]..

Table S1: Ops vocabulary (continued)

Op	Code	Input Args		Output Args		Description	
ID	Example	Addresses / types	Consts	Address / type	Index	(see caption)	
OP61	`m0=uniform(-0.5,0.6)`	–	α, β	a / matrix	–	$M_a^{(i,j)} = \mathcal{U}(\alpha, \beta)\ \forall i, j$	Prob. and Stats.
OP62	`s4=gaussian(0.1,0.7)`	–	μ, σ	a / scalar	–	$s_a = \mathcal{N}(\mu, \sigma)$	
OP63	`v8=gaussian(0.4,1)`	–	μ, σ	a / vector	–	$\vec{v}_a^{(i)} = \mathcal{N}(\mu, \sigma)\ \forall i$	
OP64	`m2=gaussian(-2,1.3)`	–	μ, σ	a / matrix	–	$M_a^{(i,j)} = \mathcal{N}(\mu, \sigma)\ \forall i, j$	

instruction removal is twice as likely as addition; (ii) completely randomize all instructions in a component function by randomizing all their ops and arguments; or (iii) modify a randomly chosen argument of a randomly selected existing instruction. All categorical random choices are uniform. When modifying a real-valued constant, we multiply it by a uniform random number in $[0.5, 2.0]$ and flip its sign with 10% probability.

We upgrade the regularized evolution search method [69] to improve its performance in the following ways. These upgrades are justified empirically through ablation studies in Supplementary Section S8.

Functional Equivalence Checking (FEC). The lack of heavy design of the search space allows for mutations that do not have an effect on the accuracy (*e.g.* adding an instruction that writes to an address that is never read). When these mutations occur, the child algorithm behaves identically to its parent. To prevent these identically functioning algorithms from being repeatedly evaluated (*i.e.* trained and validated in full many times), we keep an LRU cache mapping evaluated algorithm fingerprints to their accuracies. Before evaluating an algorithm, we first quickly fingerprint it and consult the cache to see if it has already been evaluated. If it has, we reuse the stored accuracy instead of computing it again. This way, we can keep the different implementations of the same algorithm for the sake of diversity: even though they produce the same accuracy now, they may behave differently upon further mutation.

To fingerprint an algorithm, we train it for 10 steps and validate it on 10 examples. The 20 resulting predictions are then truncated and hashed to produce an integer fingerprint. The cache holds 100k fingerprint–accuracy pairs.

Parallelism. In multi-process experiments, each process runs regularized evolution on its own population and the worker processes exchange algorithms through migration [1]. Every 100–10000 evaluations, each worker uploads 50 algorithms (half the population) to a central server. The server replies with 50 algorithms sampled randomly across *all* workers that are substituted into the worker's population.

Dataset Diversity. While the final evaluations are on binary CIFAR-10, in the experiments in Sections 4.2 and 4.3, 50% of the workers train and evaluate on binary MNIST instead of CIFAR-10. MNIST is a dataset of labeled hand-written digits [40]. We project MNIST to 256 dimensions in the same way we do for CIFAR-10. Supplementary Section S8 demonstrates how searching on multiple MNIST-based and CIFAR-based tasks improves final performance on CIFAR-10, relative to searching only on multiple MNIST-based tasks or only on multiple CIFAR-based tasks.

Hurdles. We adopt the *hurdles* upgrade to the evolutionary algorithm. This upgrade uses statistics of the population to early-stop the training of low performing models [78]. The early-stopping criterion is the failure to reach a minimum accuracy—the *hurdle*. We alter the original implementation by setting the hurdle to the 75^{th} percentile of unique accuracies of the evolving population on a rolling basis (as opposed to the stationary value used in the original implementation). This alteration gives us more predictability over the resource savings: we consistently save 75% of our compute, regardless of how the accuracy distribution shifts over the course of the search experiment.

Terminating Degenerate Algorithms. We terminate algorithms early if their calculations produce NaN or Inf values, and assign them a fixed minimum accuracy a_{min} (we use a_{min}=0). Similarly, if an algorithm's error on any training example exceeds a threshold e_{max}≫1 (we use e_{max}=100), we also stop that algorithm and assign it the accuracy a_{min}. Lastly, we time each algorithm as it is being executed and terminate it if its run-time exceeds a fixed threshold; we set this threshold to 4x the run-time of a plain neural network trained with gradient descent.

The experiment's meta-parameters (*e.g.* P and T) were either decided in smaller experiments (*e.g.* P), taken from previous work (*e.g.* T), or not tuned. Even when tuning parameters in smaller experiments, this was not done extensively (*e.g.* no multi-parameter grid searches); typically, we tried a handful of values independently when each feature was introduced. For each experiment, we scaled—without tuning—some meta-parameters based on compute or hardware limitations. For example, compute-heavy tasks use smaller populations in order to save frequent checkpoints in case of machine reboots. Additional discrepancies between experiment configurations in the different sections are due to different researchers working independently.

S3. Task Generation Details

Sections 4.2 and 4.3 employ many binary classification tasks grouped into two sets, $\mathcal{T}_{search}$ and $\mathcal{T}_{select}$. We now describe how these tasks are generated. We construct a binary classification task by randomly selecting a pair of classes from CIFAR-10 to yield positive and negative examples. We then create a random projection matrix by drawing from a Gaussian distribution with zero mean and unit variance. We use the matrix to project the features of all the examples corresponding to the class pair to a lower dimension (*i.e.* from the original 3072 to, for example, 16). The projected features are then standardized. This generates a proxy task that requires much less compute than the non-projected version. Each class pair and random projection matrix produce a different task. Since CIFAR-10 has 10 classes, there are 45 different pairs. For each pair we perform 100 different projections. This way we end up with 4500 tasks, each containing 8000/2000 training/validation examples. We use all the tasks from 36 of the pairs to form the $\mathcal{T}_{search}$ task set. The remaining tasks form $\mathcal{T}_{select}$.

S4. Detailed Search Experiment Setups

Here we present details and method meta-parameters for experiments referenced in Section 4. These complement the "Experiment Details" paragraphs in the main text.

Experiments in Section 4.1, Figure 4: Scalar/vector/matrix number of addresses: 4/3/1 (linear), 5/3/1 (affine). Fixed num. instructions for `Setup`/`Predict`/`Learn`: 5/1/4 (linear), 6/2/6 (affine). Expts. in this figure allow only minimal ops to discover a known algorithm, as follows. For "linear backprop" expts.: allowed `Learn` ops are {OP3, OP4, OP19, OP24}. For "linear regressor" expts.: allowed `Setup` ops are {OP56, OP57}, allowed predict ops are {OP27}, and allowed `Learn` ops are {OP2, OP3, OP18, OP23}. For "affine backprop" expts.: allowed `Learn` ops are {OP1, OP2, OP3, OP18, OP23}. For "affine regressor" expts.: allowed `Setup` ops are {OP56, OP57}, allowed `Predict` ops are {OP1, OP27}, and allowed `Learn` ops are {OP1, OP2, OP3, OP18, OP23}. 1 process, no server. Tasks: see *Experiment Details* paragraph in main text. Evolution expts.: P=1000; T=10; U=0.9; we initialize the population with random programs; evals. per expt. for points in plot (left to right): 10k, 10k, 10k, 100k (optimized for each problem difficulty to nearest factor of 10). Random search expts.: same num. memory addresses, same component function sizes, same total number of evaluations. These experiments are intended to be as simple as possible, so we do not use hurdles or additional data.

Experiment in Section 4.1, Figure 5: Scalar/vector/matrix number of addresses: 4/8/2. Fixed num. instructions for `Setup`/`Predict`/`Learn`: 21/3/9. In this figure, we only allow as ops those that appear in a two-layer neural network with gradient descent: allowed `Setup` ops are {OP56, OP63, OP64}, allowed `Predict` ops are {OP27, OP31, OP48}, and allowed `Learn` ops are {OP2, OP3, OP16, OP18, OP23, OP25, OP28, OP40}. Tasks: see *Experiment Details* paragraph in main text. P=1000. T=10. U=0.9. $W=1$k. Worker processes are uniformly divided into 4 groups, using parameters $T/D/P$ covering ranges in a log scale, as follows: 100k/100/100, 100k/22/215, 10k/5/464, and 100/1/1000. Uses FEC. We initialize the population with random programs.

Experiments in Section 4.2: Scalar/vector/matrix number of addresses: 8/14/3. Maximum num. instructions for `Setup`/`Predict`/`Learn`: 21/21/45. All the initialization ops are now allowed for `Setup`: {OP56, OP57, OP58, OP59, OP60, OP61, OP62, OP63, OP64}. `Predict` and `Learn` use a longer list of 58 allowed ops: {OP0, OP1, OP2, OP3, OP4, OP5, OP6, OP7, OP8, OP9, OP10, OP11, OP12, OP13, OP14, OP15, OP16, OP17, OP18, OP19, OP20, OP21, OP22, OP23, OP24, OP25, OP26, OP27, OP28, OP29, OP30, OP31, OP32, OP33, OP34, OP35, OP36, OP37, OP38, OP39, OP40, OP41, OP42, OP43, OP44, OP45, OP46, OP47, OP48, OP49, OP50, OP51, OP52, OP53, OP54, OP55, OP60, OP61}—*all* these ops are available to *both* `Predict` and `Learn`. We use all the optimizations described in Section 5, incl. additional projected binary MNIST data. Worker processes are uniformly divided to perform each possible combination of tasks: $\{$*projected binary CIFAR-10*, *projected binary MNIST*$\}$ $\otimes$ $\{N{=}800 \text{ \& } E{=}1, N{=}8000 \text{ \& } E{=}1, N{=}800 \text{ \& } E{=}10\}$ $\otimes$ $\{D{=}1, D{=}10\}$ $\otimes$ $\{F{=}8, F{=}16, F{=}256\}$; where N is the number of training examples, E is the number of training epochs, and other quantities are defined in Section 3. $P=100$. $T=10$. $U=0.9$. $W=10$k. We initialize the population with empty programs.

Experiments in Section 4.3: Scalar/vector/matrix number of addresses: 10/16/4. Maximum num. instructions for `Setup`/`Predict`/`Learn`: 21/21/45. Allowed ops for `Setup` are {OP56, OP57, OP58, OP59, OP60, OP61, OP62, OP63, OP64}, allowed ops for `Predict` and `Learn` are {OP0, OP1, OP2, OP3, OP4, OP5, OP6, OP7, OP8, OP9, OP10, OP11, OP12, OP13, OP14, OP15, OP16, OP17, OP18, OP19, OP20, OP21, OP22, OP23, OP24, OP25, OP26, OP27, OP28, OP29, OP30, OP31, OP32, OP33, OP34, OP35, OP36, OP37, OP38, OP39, OP40, OP41, OP42, OP43, OP44, OP45, OP46, OP47, OP48, OP49, OP50, OP51, OP52, OP53, OP54, OP55, OP63, OP64}. These are the same ops as in the paragraph above, except for the minor accidental replacement of uniform for Gaussian initialization ops. We use FEC and hurdles. Workers use binary CIFAR-10 dataset projected to dimension 16. Half of the workers use $D=10$ (for faster evolution), and the other half use D=100 (for more accurate evaluation). P=100. T=10. U=0.9. Section 4.3 considers three different task types: (1) In the "few training examples" task type (Figure 7a), experiments train each algorithm on 80 examples for 100 epochs for the experiments, while controls train on 800 examples for 100 epochs. (2) In the "fast training" task type (Figure 7b), experiments train on 800 examples for 10 epochs, while controls train on 800 examples for 100 epochs. (3) In the "multiple classes" task type (Figure 7c), experiments evaluate on projected 10-class CIFAR-10 classification tasks, while controls evaluate on the projected binary CIFAR-10 classification tasks described before. The 10-class tasks are generated similarly to the binary tasks, as follows. Each task contains 45K/5K training/validation examples. Each example is a CIFAR-10 image projected to 16 dimensions using a random matrix drawn from a Gaussian distribution with zero mean and unit variance. This projection matrix remains fixed for all examples within a task. The data are standardized after the projection. We use 1000 different random projection matrices to create 1000 different tasks. 80% of these tasks constitute $\mathcal{T}_{search}$ and the rest form $\mathcal{T}_{select}$. Since Section 4.2 showed that we can discover reasonable models from scratch, in Section 4.3, we initialize the population with the simple two-layer neural network with gradient descent of Figure 5 in order to save compute.

S5. Evolved Algorithms

In this section, we show the raw code for algorithms discovered by evolutionary search in Sections 4.2 and 4.3. The code in those sections was simplified and therefore has superficial differences with the corresponding code here.

Supplementary Figure S1a shows the raw code for the best evolved algorithm in Section 4.2. For comparison, Figure S1b shows the effect of removing redundant instructions through automated static analysis (details in Supplementary Section S7). For example, the instruction `v3 = gaussian(0.7,0.4)` has been deleted this way.

Finally, the fully simplified version is in the bottom right corner of Figure 6 in the main text. To achieve this simplification, we used ablation studies to find instructions that can be removed or reordered. More details can be found in Supplementary Section S7. For example, in going from Supplementary Figure S1b to Figure 6, we removed `s5 = sin(s4)` because its deletion does not significantly alter the accuracy. We also consistently renamed some variables (of course, this has no effect on the execution of the code).

```
def Setup():
  s4 = uniform(0.6, 0.2)
  v3 = gaussian(0.7, 0.4)
  v12 = gaussian(0.2, 0.6)
  s1 = uniform(-0.1, -0.2)
def Predict():
  v1 = v0 - v9
  v5 = v0 + v9
  v6 = dot(m1, v5)
  m1 = s2 * m2
  s1 = dot(v6, v1)
  s6 = cos(s4)
def Learn():
  s4 = s0 - s1
  s3 = abs(s1)
  m1 = outer(v1, v0)
  s5 = sin(s4)
  s2 = norm(m1)
  s7 = s5 / s2
  s4 = s4 + s6
  v11 = s7 * v1
  m1 = heaviside(m2)
  m1 = outer(v11, v5)
  m0 = m1 + m0
  v9 = uniform(2e-3, 0.7)
  s7 = log(s0)
  s4 = std(m0)
  m2 = m2 + m0
  m1 = s4 * m0
```

(a)

```
def Setup():
def Predict():
  v1 = v0 - v9
  v5 = v0 + v9
  v6 = dot(m1, v5)
  m1 = s2 * m2
  s1 = dot(v6, v1)
def Learn():
  s4 = s0 - s1
  m1 = outer(v1, v0)
  s5 = sin(s4)
  s2 = norm(m1)
  s7 = s5 / s2
  v11 = s7 * v1
  m1 = outer(v11, v5)
  m0 = m1 + m0
  v9 = uniform(2e-3, 0.7)
  s4 = std(m0)
  m2 = m2 + m0
  m1 = s4 * m0
```

(b)

Figure S1: (a) Raw code for the best evolved algorithm in Figure 6 (bottom–right corner) in Section 4.2. (b) Same code after redundant instructions have been removed through static analysis.

Supplementary Figure S2 shows the raw code for the algorithms in Figure 7 in Section 4.3. Note that in Figure 7, we display a code snippet containing only a few selected instructions, while Supplementary Figure S2 shows the programs in full.

S6. Algorithm Selection and Evaluation

We first run search experiments evaluating algorithms on the projected binary classification tasks sampled from $\mathcal{T}_{search}$ and collect the best performing candidate from each experiment. The measure of performance is the median accuracy across tasks. Then, we rank these candidates by evaluating them on tasks sampled from $\mathcal{T}_{select}$ and we select the highest-ranking candidate (this is analogous to typical model selection practice using a validation set). The highest ranking algorithm is finally evaluated on the binary classification tasks using CIFAR-10 data with the original dimensionality (3072).

Because the algorithms are initially evolved on tasks with low dimensionality (16) and finally evaluated on the full-size dimensionality (3072), their hyperparameters must be tuned

```
def Setup():
  s3 = 4.0e-3
def Predict():
  v7 = v5 - v0
  v3 = dot(m0, v0)
  s8 = s9 / s3
  v3 = v3 + v1
  m1 = heaviside(m3)
  v4 = maximum(v3, v7)
  s1 = dot(v4, v2)
  s4 = log(s9)
  v3 = bcast(s8)
  s7 = std(v1)
  v3 = v14 + v0
  m2 = matmul(m0, m2)
def Learn():
  v1 = gaussian(-0.50, 0.41)
  s4 = std(m0)
  s4 = s0 - s1
  v5 = gaussian(-0.48, 0.48)
  m1 = transpose(m3)
  s4 = s3 * s4
  v15 = v15 * v6
  v6 = s4 * v4
  v2 = v2 + v6
  v7 = s4 * v2
  v8 = heaviside(v3)
  v7 = v8 * v7
  m1 = outer(v7, v0)
  m0 = m0 + m1
```

(a) Raw code for the adaptation to few examples in Figure 7a.

```
def Setup():
  s3 = 0.37
  s1 = uniform(0.42, 0.66)
  s2 = 0.31
  v13 = gaussian(0.69, 0.61)
  v1 = gaussian(-0.86, 0.97)
def Predict():
  m3 = m1 + m2
  s6 = arccos(s0)
  v3 = dot(m0, v0)
  v3 = v3 - v0
  v11 = v2 + v9
  m2 = m0 - m2
  s4 = maximum(s8, s0)
  s7 = 1 / s6
  s6 = arctan(s0)
  s8 = minimum(s3, s1)
  v4 = maximum(v3, v10)
  s1 = dot(v4, v2)
  s1 = s1 + s2
def Learn():
  v1 = dot(m0, v5)
  s4 = s0 - s1
  s4 = s3 * s4
  s2 = s2 + s4
  m3 = matmul(m0, m3)
  m2 = bcast(v2, axis=0)
  v13 = s4 * v4
  v15 = v10 + v12
  v2 = v11 + v13
  v7 = s4 + v11
  v11 = v7 + v8
  s8 = s9 + s1
  m2 = m3 * m0
  s3 = arctan(s3)
  v8 = heaviside(v3)
  m2 = transpose(m1)
  s8 = heaviside(s6)
  s8 = norm(m3)
  v7 = v8 * v7
  m3 = outer(v7, v0)
  m0 = m0 + m3
```

(b) Raw code for the adaptation to fast training in Figure 7b.

```
def Setup():
  m3 = uniform(0.05, 0.11)
  s1 = uniform(0.31, 0.90)
  v18 = uniform(-0.49, 4.41)
  s1 = -0.65
  m5 = uniform(0.21, 0.22)
  v9 = gaussian(0.64, 7.8e-3)
  s1 = -0.84
def Predict():
  s1 = abs(s1)
  v15 = norm(m1, axis=1)
  v15 = dot(m0, v0)
  v8 = v19 - v0
  v15 = v8 + v15
  v7 = max(v1, v15)
  v13 = min(v5, v4)
  m2 = transpose(m2)
  v10 = v13 * v0
  m7 = heaviside(m3)
  m4 = transpose(m7)
  v2 = dot(m1, v7)
  v6 = max(v2, v9)
  v2 = v2 + v13
  v11 = heaviside(v17)
  s1 = sin(s1)
  m3 = m6 - m5
  v19 = heaviside(v14)
  v10 = min(v12, v7)
def Learn():
  m5 = abs(m7)
  v8 = v1 - v2
  m2 = transpose(m2)
  v8 = s1 * v8
  v15 = v15 - v4
  v4 = v4 + v8
  s1 = arcsin(s1)
  v15 = mean(m3, axis=1)
  v12 = v11 * v11
  m4 = heaviside(m5)
  m6 = outer(v8, v7)
  s1 = sin(s1)
  s1 = exp(s0)
  m1 = m1 + m3
  m5 = outer(v15, v6)
  m2 = transpose(m1)
  s1 = exp(s0)
  v12 = uniform(0.30, 0.33)
  s1 = minimum(s0, s1)
  m5 = m5 * m7
  v9 = dot(m2, v8)
  v9 = v10 * v9
  v3 = norm(m7, axis=1)
  s1 = mean(m1)
  m2 = outer(v9, v0)
  m0 = m0 + m2
```

(c) Raw code for the adaptation to multiple classes in Figure 7c.

Figure S2: Raw evolved code for algorithm snippets in Figure 7 in Section 4.3.

on the full-size dimensionality before that final evaluation. To do this, we treat all the constants in the algorithms as hyperparameters and jointly tune them using random search. For each random search trial, each constant is scaled up or down by a random factor sampled between 0.001 and 1000 on a log-scale. We allowed up to 10k trials to tune the hyperparameters, but only a few hundred were required to tune the best algorithm in Figure 6—note that this algorithm only has 3 constants. To make comparisons with baselines fair, we tune these baselines using the same amount of resources that went into tunining *and evolving* our algorithms. All hyperparameter-tuning trials use 8000 training and 2000 validation examples from the CIFAR-10 *training* set. After tuning, we finally run the tuned algorithms on 2000 examples from the held-out CIFAR-10 *test* set. We repeat this final evaluation with 5 different random seeds and report the mean and standard deviation. We stress that the CIFAR-10 test set was used only in this final evaluation, and never in $\mathcal{T}_{search}$ or $\mathcal{T}_{select}$.

In our experiments, we found a *hyperparameter coupling* phenomenon that hinders algorithm selection and tuning. ML algorithms usually make use of hyperparameters (*e.g*. learning rate) that need to be tuned for different datasets (for example, when the datasets have very different input dimensions or numbers of examples). Similarly, the evolved algorithms also contain hyperparameters that need to be adjusted for different datasets. If the hyperparameters are represented as constants in the evolved algorithm, we can identify and tune them on the new dataset by using random search. However, it is harder to tune them if a hyperparameter is instead computed from other variables. For example, in some evolved algorithms, the learning rate s_2 was computed as $s_2 = norm(v_1)$ because the best value for s_2 coincides with the L2-norm of v_1 on $\mathcal{T}_{search}$. However, when we move to a new dataset with higher dimensions, the L2-norm of v_1 might no longer be a good learning rate. This can cause the evolved algorithms' performance to drop dramatically on the new dataset. To resolve this, we identify these parameters by manual inspection of the evolved code. We then manually decouple them: in the example, we would set s_2 to a constant that we can tune with random-search. This recovers the performance. Automating the decoupling process would be a useful direction for future work.

S7. Interpreting Algorithms

It is nontrivial to interpret the raw evolved code and decide which sections of it are important. We use the following procedures to help with the interpretation of discovered algorithms:

(1) We clean up the raw code (*e.g*. Figure S1a) by automatically simplifying programs. To do this, we remove redundant instructions through static analysis, resulting in code like that in Figure S1b. Namely, we analyze the computations that lead to the final prediction and remove instructions that have no effect. For example, we remove instructions that initialize variables that are never used.

(2) We focus our attention on code sections that reappear in many independent search experiments. This is a sign that such code sections may be beneficial. For example, Section 4.3 applied this procedure to identify adaptations to different tasks.

(3) Once we have hypotheses about interesting code sections, we perform ablations/*knock-outs*, where we remove the code section from the algorithm to see if there is a significant loss in accuracy. As an example, for Section 4.2, we identified 6 interesting code sections in the best evolved algorithm to perform ablations. For each ablation, we removed the relevant code section, then tuned all the hyperparameters / constants again, and then computed the loss in validation accuracy. 4 out of the 6 ablations caused a large drop in accuracy. These are the ones that we discussed in Section 4.2. Namely, (1) the addition of noise to the input (-0.16%); (2) the bilinear model (-1.46%); (3) the normalized gradients (-1.20%); and (4) the weight averaging (-4.11%). The remaining 2 code sections show no significant loss upon ablation, and so were removed for code readability. Also for readability, we reorder some instructions when this makes no difference to the accuracy either (*e.g*. we move related code lines closer to each other). After this procedure, the code looks like that in Figure 6.

(4) If an ablation suggests that a code section is indeed helpful to the original algorithm, we then perform a *knock-in*. That is, we insert the code section into simpler algorithms to see if it improves their performance too. This way we confirmed the usefulness of the 4 code sections mentioned above, for example.

S8. More Search Method Ablations

To verify the effectiveness of the upgrades mentioned in Supplementary Section S2, we conduct ablation studies using the experiment setup of Section 4.2, except for the following simplifications to reduce compute: (i) we limit the ops to only those that are necessary to construct a neural network trained with gradient descent (as was done for Figure 5), *i.e.* allowed `Setup` ops are {OP57, OP64, OP65}, allowed `Predict` ops are {OP28, OP32, OP49}, and allowed `Learn` ops are {OP3, OP4, OP17, OP19, OP24, OP26, OP29, OP41}; (ii) we reduce the projected dimensionality from 256 to 16, and (iii) we use 1k processes for 5 days. Additionally, the ablation experiments we present use $T = 8000, E = 10$ for all tasks. This slight difference is not intentionally introduced, but rather is a product of our having studied our method before running our experiments in the main text; we later on found that using more epochs did not yield more information or improve the results.

Supplementary Tables S2, S3, and S4 display the results. Note that Figure 8 presents a subset of this data in plot form (the indexes 1–4 along the horizontal axis labels in that figure coincide with the "Index" column in this table). We find that all four upgrades are beneficial across the three different compute scales tested.

Table S2: Ablation studies. Each row summarizes the results of 30 search runs under one given experimental setting. Rows #0–4 all use the same setting, except for the search method: each row implements an upgrade to the method and shows the resulting improvement. “Best Accuracy” is the accuracy of the best algorithm for each experiment ($\pm 2\,\mathrm{SEM}$), evaluated on unseen projected binary CIFAR-10 tasks. “Success Fraction” is the fraction ($\pm 2\sigma$) of those experiments that produce algorithms that are more accurate than a plain neural network trained with gradient descent (0.750). This fraction helps us estimate the likelihood of high performing outliers, which we are keenly interested in. The experimental setting for row #5 is the same as row #3, except that instead of using both projected binary CIFAR-10 and projected binary MNIST data for the search, we use only projected binary MNIST data (as for other rows, the accuracy is reported on projected binary CIFAR-10 data). Row #5 indicates that searching *completely* on MNIST data is not as helpful as searching partially on it (row #3). Overall, rows #0–4 suggest that all four upgrades are beneficial.

Index	Description	Best Accuracy	Success Fraction
0	Baseline	0.703 ± 0.002	0.00 ± 0.00
1	+ Migration	0.707 ± 0.004	0.00 ± 0.00
2	+ Functional Equivalence Check	0.724 ± 0.006	0.13 ± 0.12
3	+ 50% MNIST Data	0.729 ± 0.008	0.27 ± 0.16
4	+ Hurdles	0.738 ± 0.008	0.53 ± 0.18
5	Experiment 3 w/ 100% MNIST Data	0.720 ± 0.003	0.00 ± 0.00

Table S3: This is the same as Supplementary Table S2, except at a lower compute scale (100 processes). Each setup was run 100 times. The results are similar to those with more compute and support the same conclusions. Thus, the observed benefits are not specific to a single compute scale.

Index	Description	Best Accuracy	Success Fraction
0	Baseline	0.700 ± 0.002	0.00 ± 0.00
1	+ Migration	0.704 ± 0.000	0.00 ± 0.00
2	+ Functional Equivalence Check	0.706 ± 0.001	0.00 ± 0.00
3	+ 50% MNIST Data	0.710 ± 0.002	0.02 ± 0.03
4	+ Hurdles	0.714 ± 0.003	0.10 ± 0.06
5	Experiment 3 w/ 100% MNIST Data	0.700 ± 0.004	0.00 ± 0.00

Table S4: This is the same as Supplementary Tables S2 and S3, except at an even lower compute scale (10 processes). Each setup was run 100 times. The results are consistent with those with more compute, but we no longer observe successes (“successes” defined in Table S2).

Index	Description	Best Accuracy	Success Fraction
0	Baseline	0.700 ± 0.001	0.00 ± 0.00
1	+ Migration	0.701 ± 0.001	0.00 ± 0.00
2	+ Functional Equivalence Check	0.702 ± 0.001	0.00 ± 0.00
3	+ 50% MNIST Data	0.704 ± 0.001	0.00 ± 0.00
4	+ Hurdles	0.705 ± 0.001	0.00 ± 0.00
5	Experiment 3 w/ 100% MNIST Data	0.694 ± 0.002	0.00 ± 0.00

S9. Baselines

The focus of this study was not the search method but we believe there is much room for future work in this regard. To facilitate comparisons with other search algorithms on the same search space, in this section we provide convenient baselines at three different compute scales.

All baselines use the same setting, a simplified version of that in Section 4.2, designed to use less compute. In particular, here we severely restrict the search space to be able to reach results quickly. Scalar/vector/matrix number of addresses: 5/9/2. Maximum num. instructions for `Setup`/`Predict`/`Learn`: 7/11/23. Allowed `Setup` ops: {OP57, OP60, OP61, OP62, OP63, OP64, OP65}, allowed `Predict` ops: {OP2, OP24, OP28, OP32, OP49}, allowed `Learn` ops: {OP2, OP3, OP4, OP17, OP19, OP24, OP26, OP29, OP41}. Experiments end after each process has run 100B training steps—*i.e.* the training loop described in Section 3.1 runs 100B times. (We chose “training steps” instead of “number of algorithms” as the experiment-ending criterion because the latter varies due to early stopping, FEC, *etc.* We also did not choose “time” as the experiment-

Table S5: Baselines on the simplified setting with the restricted search space (see Supplementary Section S9 for details). "Best Accuracy" is the best evaluated accuracy on unseen projected binary classification tasks for each run ($\pm 2\,\mathrm{SEM}$), "Linear Success Fraction" is the fraction ($\pm 2\sigma$) of those accuracies that are above the evaluated accuracy of logistic regression trained with gradient descent (0.702), and "NN Success Fraction" is the fraction ($\pm 2\sigma$) of those accuracies that are above the evaluated accuracy of a plain neural network trained with gradient descent (0.729). Using success fractions as a metric helps us estimate the likelihood of discovering high performing outliers, which we are keenly interested in. Each experiment setup was run 100 times. The "Full" method is the one we used in Section 4.2; the "Basic" method is the same, but with no FEC, no hurdles, and no MNIST data.

METHOD	NUMBER OF PROCESSES	BEST ACCURACY	LINEAR SUCCESS FRACTION	NN SUCCESS FRACTION
BASIC	1	0.671 ± 0.004	0.01 ± 0.02	0.00 ± 0.00
BASIC	10	0.681 ± 0.005	0.07 ± 0.05	0.00 ± 0.00
BASIC	100	0.691 ± 0.004	0.26 ± 0.09	0.00 ± 0.00
FULL	1	0.684 ± 0.003	0.03 ± 0.03	0.00 ± 0.00
FULL	10	0.693 ± 0.003	0.23 ± 0.08	0.03 ± 0.03
FULL	100	0.707 ± 0.003	0.59 ± 0.10	0.11 ± 0.06

ending criterion to make comparisons hardware-agnostic. For reference, each experiment took roughly 12 hours on our hardware.) P=100; T=10, U=0.9. All workers evaluate on the same projected binary CIFAR-10 tasks as in Section 4.2, except that we project to 16 dimensions instead of 256. Each search evaluation is on 10 tasks and the numbers we present here are evaluations on $\mathcal{T}_{select}$ using 100 tasks. We initialize the population with empty programs.

The results of performing 100 repeats of these experiments at three different compute scales are summarized in Table S5. We additionally compare our full search method from Section 4.2, labeled "Full", and a more "Basic" search setup, which does not use FEC, hurdles, or MNIST data.