# AutoML-Zero: Evolving Machine Learning Algorithms From Scratch

# Supplementary Material

## S1. Additional 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 [Fahlman & Lebiere, 1990; Angeline et al., 1994; Yao, 1999; Stanley & Miikkulainen, 2002; Bergstra & Bengio, 2012; Mendoza et al., 2016; Baker et al., 2017; Zoph & Le, 2016; Real et al., 2017; Xie & Yuille, 2017; Suganuma et al., 2017; Liu et al., 2018, and many others] and continues today [Liu et al., 2019b; Elsken et al., 2019a; Cai et al., 2019; Liu et al., 2019a; Ghiasi et al., 2019; Sun et al., 2019; Xie et al., 2019, and many others]. Reviews provide more thorough background (Elsken et al., 2019b; Stanley et al., 2019; Yao et al., 2018). Recent works have 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 (Zoph et al., 2018; Zhong et al., 2018). Mei et al. (2020) 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 (Snoek et al., 2012; Loshchilov & Hutter, 2016; Jaderberg et al., 2017; Li et al., 2018), activation functions (Ramachandran et al., 2017), a specific layer (Kim & Rigazio, 2015), the full forward pass (Gaier & Ha, 2019), the data augmentation (Cubuk et al., 2019a; Park et al., 2019; Cubuk et al., 2019b), *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 (Miikkulainen et al., 2019), the architecture and the hyperparameters (Zela et al., 2018), the layers and the weight pruning (Noy et al., 2019), and so on. We extend this to all aspects of the algorithm, including the optimization.

An important aspect of an ML algorithm is optimization, which has been tackled by AutoML in the form of numerically discovered optimizers. Chalmers (1991) 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 structures, such as a second neural network (Runarsson & Jonsson, 2000; Orchard & Wang, 2016), an LSTM (Ravi & Larochelle, 2017), a hierarchical RNN (Wichrowska et al., 2017), or even a different LSTM for each weight (Metz et al., 2019). 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 (Andrychowicz et al., 2016), to reinforcement learning (Li & Malik, 2017), to evolutionary search with sophisticated developmental encodings (Risi & Stanley, 2010). All these methods are sometimes collectively labeled as *meta-learning* (Vanschoren, 2019) 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 also the structure and the initialization of the model. 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. (1994), 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* (Holland, 1975; Forsyth et al., 1981; Koza & Koza, 1992). 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. (1994) 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. (2017) 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. (1994) and Bello et al. (2017) 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 neural network model or of the gradient. They must therefore be discovered in the same way as the rest of the algorithm.

We note that our work also relates to program synthesis efforts. Early approaches have proposed to search for programs that improve themselves (Lenat, 1983; Schmidhuber, 1987; Pitrat, 1996). We share similar goals in searching for learning algorithms, but focus on common machine learning tasks and have dropped the self-reflexivity requirement. More recently, program synthesis has focused on solving problems like sorting, addition, counting (Schmidhuber, 2004; Graves et al., 2014; Reed & de Freitas, 2015; Valkov et al., 2018), string manipulations (Polozov & Gulwani, 2015; Parisotto et al., 2016; Devlin et al., 2017), character recognition (Lake et al., 2015), competition-style programming (Balog et al., 2017), structured data QA (Neelakantan et al., 2015; Liang et al., 2016; 2018), program parsing (Chen et al., 2017), and game playing (Wilson et al., 2018), to name a few. These studies are increasingly making more *use* of ML to solved the said problems (Gulwani et al., 2017). Unlike these studies, we focus on synthesizing programs that solve the problem of *doing* ML.

## S2. 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.

## S3. 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, 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 (Real et al., 2019) to improve its performance in the following ways. These upgrades are justified empirically through ablation studies in Supplementary Section S9.

**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 (Alba & Tomassini, 2002). 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 (LeCun et al., 1998). We project MNIST to 256 dimensions in the same way we do for CIFAR-10. Supplementary Section S9 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 (So et al., 2019). 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 $\mathrm{NaN}$ or $\mathrm{Inf}$ 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 $e_{max}{\gg}1$ (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

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 $\mu$ and standard deviation $\sigma$. $\mathbb{1}_X$ 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 ID | Code Example | Input Args | | Output Args | | Description (see caption) | |
|---|---|---|---|---|---|---|---|
| | | Addresses / types | Consts. | Address / type | Index | | |
| 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 (see caption) | |
|---|---|---|---|---|---|---|---|
| | | Addresses / types | Consts | Address / type | Index | | |
| 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 | – | $s_b = \lvert\lvert M_a \rvert\rvert$ | |
| OP35 | `v4=norm(m7,axis=0)` | *a* / matrix | – | *b* / vector | – | $\vec{v}_b^{(i)} = \lvert M_a^{(i,\cdot)} \rvert\ \forall i$ | |
| OP36 | `v4=norm(m7,axis=1)` | *a* / matrix | – | *b* / vector | – | $\vec{v}_b^{(j)} = \lvert M_a^{(\cdot,j)} \rvert\ \forall j$ | |
| OP37 | `m9=transpose(m3)` | *a* / matrix | – | *b* / matrix | – | $M_b = \lvert M_a^T \rvert$ | |
| OP38 | `m1=abs(m8)` | *a* / matrix | – | *b* / matrix | – | $M_b^{(i,j)} = \lvert M_a^{(i,j)} \rvert\ \forall i,j$ | |
| 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` | – | $\gamma$ | *a* / scalar | – | $s_a = \gamma$ | |
| OP57 | `v3[5]=-2.4` | – | $\gamma$ | *a* / vector | $i$ | $\vec{v}_a^{(i)} = \gamma$ | |
| OP58 | `m2[5,1]=-0.03` | – | $\gamma$ | *a* / matrix | $i, j$ | $M_a^{(i,j)} = \gamma$ | |
| OP59 | `s4=uniform(-1,1)` | – | $\alpha, \beta$ | *a* / scalar | – | $s_a = \mathcal{U}(\alpha, \beta)$ | |
| OP60 | `v1=uniform(0.4,0.8)` | – | $\alpha, \beta$ | *a* / vector | – | $\vec{v}_a^{(i)} = \mathcal{U}(\alpha, \beta)\ \forall i$ | |

[Table continues on the next page.]

Table S1: Ops vocabulary (continued)

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

case of machine reboots. Additional discrepancies between experiment configurations in the different sections are due to different researchers working independently.

## S4. 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 $\mathcal{T}_{select}$.

## S5. 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. $W$=1k. 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*$\}$ $\otimes$ $\{N{=}800$ & $E{=}1, N{=}8000$ & $E{=}1, N{=}800$ & $E{=}10\}$ $\otimes$ $\{D{=}1, D{=}10\}$ $\otimes$ $\{F{=}8, F{=}16, F{=}256\}$; 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 processes (commodity CPU cores). 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.

## S6. 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 S8). For example, the instruction v3 = gaussian(0.7,0.4) has been deleted this way.

```
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.

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 S8. 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():
  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.

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.

## S7. 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 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 $10$k 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.

## S8. 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:

(a) 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.

(b) 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.

(c) 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.

(d) 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 in (c), for example.

## S9. More Search Method Ablations

To verify the effectiveness of the upgrades mentioned in Supplementary Section S3, 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

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 ($\pm 2\,\text{SEM}$), 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 | $0.703 \pm 0.002$ | $0.00 \pm 0.00$ |
| 1 | + Migration | $0.707 \pm 0.004$ | $0.00 \pm 0.00$ |
| 2 | + Functional Equivalence Check | $0.724 \pm 0.006$ | $0.13 \pm 0.12$ |
| 3 | + 50% MNIST Data | $0.729 \pm 0.008$ | $0.27 \pm 0.16$ |
| 4 | + Hurdles | $0.738 \pm 0.008$ | $0.53 \pm 0.18$ |
| 5 | Experiment 3 w/ 100% MNIST Data | $0.720 \pm 0.003$ | $0.00 \pm 0.00$ |

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 | $0.700 \pm 0.002$ | $0.00 \pm 0.00$ |
| 1 | + Migration | $0.704 \pm 0.000$ | $0.00 \pm 0.00$ |
| 2 | + Functional Equivalence Check | $0.706 \pm 0.001$ | $0.00 \pm 0.00$ |
| 3 | + 50% MNIST Data | $0.710 \pm 0.002$ | $0.02 \pm 0.03$ |
| 4 | + Hurdles | $0.714 \pm 0.003$ | $0.10 \pm 0.06$ |
| 5 | Experiment 3 w/ 100% MNIST Data | $0.700 \pm 0.004$ | $0.00 \pm 0.00$ |

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 change the conclusions of the studies 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.

## S10. 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-ending criterion to make comparisons hardware-agnostic.

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 | $0.700 \pm 0.001$ | $0.00 \pm 0.00$ |
| 1 | + Migration | $0.701 \pm 0.001$ | $0.00 \pm 0.00$ |
| 2 | + Functional Equivalence Check | $0.702 \pm 0.001$ | $0.00 \pm 0.00$ |
| 3 | + 50% MNIST Data | $0.704 \pm 0.001$ | $0.00 \pm 0.00$ |
| 4 | + Hurdles | $0.705 \pm 0.001$ | $0.00 \pm 0.00$ |
| 5 | Experiment 3 w/ 100% MNIST Data | $0.694 \pm 0.002$ | $0.00 \pm 0.00$ |

Table S5: Baselines on the simplified setting with the restricted search space (see Supplementary Section S10 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 | $0.671 \pm 0.004$ | $0.01 \pm 0.02$ | $0.00 \pm 0.00$ |
| Basic | 10 | $0.681 \pm 0.005$ | $0.07 \pm 0.05$ | $0.00 \pm 0.00$ |
| Basic | 100 | $0.691 \pm 0.004$ | $0.26 \pm 0.09$ | $0.00 \pm 0.00$ |
| Full | 1 | $0.684 \pm 0.003$ | $0.03 \pm 0.03$ | $0.00 \pm 0.00$ |
| Full | 10 | $0.693 \pm 0.003$ | $0.23 \pm 0.08$ | $0.03 \pm 0.03$ |
| Full | 100 | $0.707 \pm 0.003$ | $0.59 \pm 0.10$ | $0.11 \pm 0.06$ |

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; note, each process is run on a single commodity CPU core. 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.