Fast Neural Models for Symbolic Regression at Scale

Allan Costa[*]
MIT
allanc@mit.edu

Rumen Dangovski[*]
MIT
rumenrd@mit.edu

Owen Dugan[*]
MIT
odugan@mit.edu

Samuel Kim
MIT
samkim@mit.edu

Pawan Goyal
MIT
pawan14@mit.edu

Marin Soljačić[†]
MIT
soljacic@mit.edu

Joseph Jacobson[†]
MIT
jacobson@media.mit.edu

Abstract

Deep learning owes much of its success to the astonishing expressiveness of neural networks. However, this comes at the cost of complex, black-boxed models that extrapolate poorly beyond the domain of the training dataset, conflicting with goals of finding analytic expressions to describe science, engineering and real world data. Under the hypothesis that the hierarchical modularity of such laws can be captured by training a neural network, we introduce OccamNet, a neural network model that finds interpretable, compact, and sparse solutions for fitting data, à la Occam's razor. Our model defines a probability distribution over a non-differentiable function space. We introduce a two-step optimization method that samples functions and updates the weights with backpropagation based on cross-entropy matching in an evolutionary strategy: we train by biasing the probability mass toward better fitting solutions. OccamNet is able to fit a variety of symbolic laws including simple analytic functions, recursive programs, implicit functions, simple image classification, and can outperform noticeably state-of-the-art symbolic regression methods on real world regression datasets. Our method requires minimal memory footprint, does not require AI accelerators for efficient training, fits complicated functions in minutes of training on a single CPU, and demonstrates significant performance gains when scaled on a GPU. Our implementation, demonstrations and instructions for reproducing the experiments are available at https://github.com/druidowm/OccamNet_Public.

1 Introduction

Deep learning has revolutionized a variety of complex tasks, ranging from language modeling to computer vision [1]. Key to this success is designing a large search space in which many local minima sufficiently approximate given data [2]. This requires large, complex models, which conflicts with the goals of sparsity and interpretability, making neural nets ill-suited for a myriad of physical and computational problems that have compact and interpretable underlying mathematical structures [3].

While neural networks easily emulate data generated from symbolic laws, the resulting models are not interpretable, might not preserve desired physical properties (e.g. time invariance), and are unable to generalize beyond observed data. Moreover, neural networks' reliance on complexity implies that reproducing compact laws might require the full collection of trained weights, in opposition to a compact form solution.

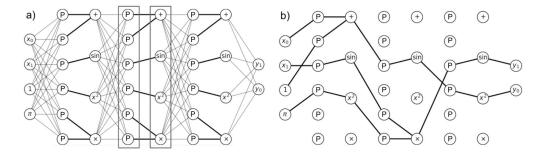[*]Equal contribution.
[†]Equal supervision.

Figure 1: (*a*) A two-output network model with depth $L = 2$, $\vec{x} = [x_0, x_1]$, user selected constants $\mathcal{C} = [1, \pi]$, and bases functions $\boldsymbol{\Phi} = \langle +(\cdot, \cdot), \sin(\cdot), (\cdot)^2, \times(\cdot, \cdot) \rangle$. Highlighted are the arguments sublayer, composed of P-nodes, and the images sublayer, composed of the bases functions from $\boldsymbol{\Phi}$. Together, these two sublayers define a single layer of our model. (*b*) An example of function-specifying directed acyclic graphs (DAGs) that can be sampled from the network in (*a*).

In contrast, Evolutionary Algorithms (EAs) have been successful in symbolic regression as they can find interpretable, compact models that explain observed data [4]. EAs have been employed as an alternative to gradient descent for optimizing neural networks, in what is called Neuroevolution [5–7]. Traditionally, these algorithms operate over populations of candidate solutions using methods inspired by biological evolution, such as mutations and selection of the fittest [4]. More recently, evolutionary strategies that model a probability distribution over parameters, updating this distribution according to their own best samples (i.e. selecting the fittest), were found advantageous in optimization on high-dimensional spaces, including neural networks' hyperparameters [8, 9]. This approach is interesting for the purposes of Neuroevolution, as keeping a probability distribution over the weights requires less storage than keeping a population of networks over which selection occurs.

In this paper, we consider a mixed approach of connectionist and evolutionary optimization for symbolic regression. We use a neural network to model a probability distribution over functions, and optimize the model through a novel two-step gradient-descent and evolutionary strategy training. We introduce a loss function that is tunable for different tasks. Our method handles non-differentiable and implicit functions, converges to sparse, interpretable symbolic expressions, and can even outperform state-of-the-art symbolic regression algorithms in testing on real world regression datasets. We also introduce a number of strategies to induce compactness and simplicity, à la Occam's Razor.

2 Model Architecture

Layer structure   A dataset $\mathcal{D} = \{\langle \vec{x}_p, \vec{y}_p \rangle\}_{p=1}^{|\mathcal{D}|}$ consists of pairs of inputs $\vec{x}_p$ and targets $\vec{y}_p = \vec{f}^*(\vec{x}_p) = [f^*_{(0)}(\vec{x}_p), \ldots, f^*_{(v-1)}(\vec{x}_p)]^\top$. Our goal is to compose either $f^*_{(i)}(\cdot)$ or an approximation of $f^*_{(i)}(\cdot)$ using a predefined collection of N basis functions $\boldsymbol{\Phi} = \{\phi_i(\cdot)\}_{i=1}^N$, which act as primitives for programs or functions. Note that bases can be repeated, their arity (number of arguments) is not restricted to one, they may operate over different domains, and they may involve unspecified constants (such as in $\phi_i(x) = x^c$). Furthermore, the concept of bases $\boldsymbol{\Phi}$ is similar to that of DSL, *domain specific languages* [10]. We concatenate the input $\vec{x}$ with a predefined collection of constants (e.g., π, 1, e) to build constant factors in our solution.

The architecture to solve this problem resembles a conventional network with L fully connected or sparse layers with no bias. To incorporate the bases $\boldsymbol{\Phi}$ in the network, we follow a similar approach as [11–13], in which the bases act as activation functions on the nodes of the network. Specifically, each hidden layer consists of an *arguments* sublayer and an *images* sublayer. The bases are stacked in the images sublayer and act as activation functions for their respective nodes. Each basis takes in nodes from the arguments sublayer. We call nodes in the arguments sublayer *P-nodes*, because they behave probabilistically, as discussed in Section 2. Figure 1 highlights this sublayer structure, while the supplemental material (SM) describes the complete mathematical formalism behind it.

Temperature-controlled connectivity To make the network more interpretable, we maximize sparsity. There are numerous approaches to inducing sparsity in neural networks, including L_1, L_0 and $L_{1/2}$ regularization [14–16], but these methods indiscriminately regularize all weights equally without capturing structure within layers. We propose a sparsity method which uses the probabilistic interpretation of the softmax function by sampling sparse paths through a network.

To promote probability-based sparsity, we use a network of T-*softmax layers*. For any temperature $T > 0$, we define a T-softmax layer as a standard T-controlled softmax layer with weighted edges connecting an images sublayer and the subsequent arguments sublayer in which each P-node from the arguments sublayer probabilistically samples a single edge between itself and a node in the images sublayer. We define $\mathbf{w}^{(l,i)}$ as the weights for edges leading to the ith P-node of the lth layer and $\mathbf{W} = \left\{ \mathbf{w}^{(l,i)}; 1 \le l \le L, 1 \le i \le N \right\}$. Each node's sampling distribution is given by $\mathbf{p}^{(l,i)}(\cdot; T) = \mathrm{softmax}(\mathbf{w}^{(l,i)}; T)$, whose limit is a delta function as $T \to 0$. Selecting these edges for all T-softmax layers produces a sparse DAG specifying a function $\vec{f}$, as seen in Figure 1b. These sampled edges are encoded as sparse matrices, through which a forward pass evaluates $\vec{f}$. The probability of the model sampling $f_{(i)}$ as its ith output, $q_i(f_{(i)}|\mathbf{W})$, is the product of the probabilities of the edges of $f_{(i)}$'s DAG. Similarly, $q(\vec{f}|\mathbf{W})$, the probability of the model sampling $\vec{f}$, is given by the product of $\vec{f}$'s edges, or $q(\vec{f}|\mathbf{W}) = \prod_{i=0}^{v-1} q_i(f_{(i)}|\mathbf{W})$. In practice, we set T to a fixed, typically small, number. The last layer is usually set to a higher temperature to allow more compositionality. Moreover, the temperature can be scheduled in the spirit of simulated annealing [17, 18], which we did not find to be crucial empirically and thus discuss further in the SM.

A neural network as a probability distribution over functions Our model represents a distribution $q(\cdot|\mathbf{W})$ over a function space of all functions sampleable by the network, $\mathcal{F}_{\mathbf{\Phi}}^L = \{$all function compositions up to nesting depth L of $\mathbf{\Phi}\}$. We can then define the optimal weights $\mathbf{W}_*$ of our neural network as satisfying $q(\vec{f}|\mathbf{W}_*) = 1$ for some $\vec{f}$ such that $\vec{f}(x) = \vec{f}^*(x)$ for all x in the domain of $\vec{f}^*$. Note that since $q(\cdot|\cdot)$ is a probability distribution we have $\sum_{\vec{f} \in \mathcal{F}_{\mathbf{\Phi}}^L} q(\vec{f}|\mathbf{W}) = 1$ and $q(\vec{f}|\mathbf{W}) \ge 0$ for all $\vec{f}$ in $\mathcal{F}_{\mathbf{\Phi}}^L$. We initialize the network with weights $\mathbf{W}_\mathrm{i}$ such that $q(\vec{f_1}|\mathbf{W}_\mathrm{i}) = q(\vec{f_2}|\mathbf{W}_\mathrm{i})$ for all $\vec{f_1}$ and $\vec{f_2}$ in $\mathcal{F}_{\mathbf{\Phi}}^L$. After training, discussed in Section 3, the network has weights $\mathbf{W}_\mathrm{f}$. The network then selects the function $\vec{f_\mathrm{f}}$ with the highest probability $q(\vec{f_\mathrm{f}}|\mathbf{W}_\mathrm{f})$. We discuss our algorithms for initialization and function selection in the SM. Note that our model can be viewed as a stochastic computational graph (SCG). SCGs with discrete stochastic nodes have been optimized using backpropagation through continuous relaxations of the discrete nodes [19–21]. In contrast, here we optimize the SCG via an evolutionary strategy, as we explain in Section 3.
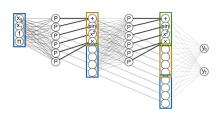

Figure 2: Skip connections. Dotted lines and colour: origin of the reused neurons.

Skip connections We use skip connections similar to those in DenseNet [22] and ResNet [23], concatenating image states with those of subsequent layers, as depicted in Figure 2. Skip connections yield several desirable properties: (*i*) The network can find compact solutions as it considers all levels of composition. This promotes solution sparsity and interpretability. (*ii*) Shallow layers are trained before or alongside the subsequent layers due to more direct supervision, because gradients can propagate to shallow layers more easily to avoid exploding or vanishing gradients. This may also allow subsequent layers to behave as higher-order corrections to the solutions found in early layers. (*iii*) Primitives in shallow layers can be reused, analogous to feature reuse in DenseNet.

3 Training

To express a wide range of functions, we include non-differentiable bases. In symbolic regression we are interested in finding the few global minima that correspond to the optimal solution. Although it is possible to incorporate known constants as inputs, it is often desireable to discover functions with unspecified constants. To address these constraints, we propose a loss function and a alternating

training method that combine gradient based optimization and evolutionary strategies for efficient global exploration of the function space. We also propose regularization terms to improve fitting.

Loss Consider a mini-batch $\mathcal{M} = \langle X, Y \rangle$, and a sampled function from the network $\vec{f}(\cdot) \sim q(\cdot | \mathbf{W})$. We compute the *fitness* of each $f_{(i)}(\cdot)$ with respect to a training pair $\langle \vec{x}, \vec{y} \rangle$ by evaluating the likelihood $k_i\left(f_{(i)}(\vec{x}), \vec{y}\right) = (2\pi\sigma^2)^{-1/2} \exp\left(-\left[f_{(i)}(\vec{x}) - (\vec{y})_i\right]^2 / (2\sigma^2)\right)$, which is a Normal distribution with mean y and variance σ^2, and measures how close $f_{(i)}(\vec{x})$ is to the target $(\vec{y})_i$. The likelihood can be also viewed as a Bayesian posterior with a noninformative prior. The total fitness is determined by summing over the entire mini-batch: $K_i\left(\mathcal{M}, f_{(i)}\right) = \sum_{(\vec{x},\vec{y}) \in \mathcal{M}} k_i\left(f_{(i)}(\vec{x}), \vec{y}\right)$.

The variance of $k_i\left(f_{(i)}(\vec{x}), \vec{y}\right)$ characterizes the fitness function's smoothness. As $\sigma^2 \to 0$, the likelihood is a delta function with nonzero fitness for some $\langle \vec{x}, \vec{y} \rangle$ only if $f_{(i)}(\vec{x}) = (\vec{y})_i$. Similarly, a large variance characterizes a fitness in which potentially many solutions provide accurate approximations, increasing the risk of convergence to local minima. In the former case, learning becomes harder as few $f_{(i)}(\cdot)$ out of exponentially many samplable functions result in any signal, whereas in the later case learning might not converge to the optimal solution. We let σ^2 be a network hyperparameter, tuned for the tradeoff between ease of learning and solution optimality for different tasks.

We now introduce a loss function for backpropagating on the weights of $q(\cdot | \mathbf{W})$:

$$H_{q_i}[f_{(i)}, \mathbf{W}, \mathcal{M}] = -K_i\left(\mathcal{M}, f_{(i)}\right) \cdot \log\left[q_i(f_{(i)} | \mathbf{W})\right]. \tag{1}$$

We can interpret (1) as the cross-entropy of the posterior for the target and the probability of the sampled function $f_{(i)}$. If the sampled function $f_{(i)}$ is close to $f_{(i)}^*$, then $K_i(\mathcal{M}, f_{(i)})$ will be large and the gradient update below, will also be large:

$$\nabla_{\mathbf{W}} H_{q_i}\left[f_{(i)}, \mathbf{W}, \mathcal{M}\right] = -\frac{\nabla_{\mathbf{W}} q_i(f_{(i)} | \mathbf{W})}{q_i(f_{(i)} | \mathbf{W})} K_i\left(\mathcal{M}, f_{(i)}\right). \tag{2}$$

The first term on the right hand side (RHS) of update (2) increases the likelihood of the function $f_{(i)}$. The second term on the RHS is maximal when $f_{(i)} \equiv f_{(i)}^*$. Importantly, the second term approaches zero as $f_{(i)}$ deviates from $f_{(i)}^*$. If the sampled function is far from the target, then the likelihood update is suppressed by $K_i(\mathcal{M}, f_{(i)})$. Therefore, we only optimize the likelihood for functions close to the target. Note that in (2) we backpropagate only through the probability of the function $f_{(i)}$ given by $q_i\left(f_{(i)} | \mathbf{W}\right)$, whose value *does not* depend on the bases in $\mathbf{\Phi}$, implying that the bases can be non-differentiable. This is particularly useful for applications requiring non-differentiable basis functions. Furthermore, this loss function allows non-differentiable regularization terms, which greatly expands the regularization possibilities.

Evolutionary strategy We find that sampling R functions in each step and performing a gradient step for each sampled function as defined in (2) easily converges to inadequate local minima. Instead, we propose an evolutionary strategy to update our model. We denote $\mathbf{W}^{(t)}$ as the set of weights at training step t, and we fix two hyperparameters: R, the number of functions to sample at each training step; and λ, or the *truncation parameter*, which defines the number of the R paths chosen for optimization via (2). We initialize $\mathbf{W}^{(0)}$ as described in Section 2. We then proceed as follows:

1. Sample R functions $\vec{f}_1, \ldots, \vec{f}_R \sim q(\cdot | \mathbf{W}^{(t)})$. We denote the jth output of $\vec{f}_i$ as $f_{i(j)}$.

2. For each output j, sort $f_{i(j)}$ from greatest to least value of $K_j\left(\mathcal{M}, f_{i(j)}\right)$ and select the top λ functions, yielding a total of $v\lambda$ selected functions $g_{1,j}, \ldots, g_{\lambda,j}$. The total loss is then given by $\sum_{i=1}^{\lambda} \sum_{j=0}^{v-1} H_{q_j}[g_{i,j}, \mathbf{W}, \mathcal{M}]$, which yields the training step gradient update:

$$-\sum_{i=1}^{\lambda} \sum_{j=0}^{v-1} \frac{\nabla_{\mathbf{W}} q_j(g_{i,j} | \mathbf{W})}{q_j(g_{i,j} | \mathbf{W})} K_j(\mathcal{M}, g_{i,j}). \tag{3}$$

   Notice that through (3) we have arrived at a modified REINFORCE update [24], where the policy is $q_i(\cdot | \cdot)$ and the regret is the fitness $K_i(\cdot, \cdot)$.

3. Perform the gradient step (3) on $\mathbf{W}^{(t)}$ for all selected paths to obtain $\mathbf{W}^{(t+1)}$. In practice, we find that the Adam algorithm [25] works well.

4

4. Set $t = t + 1$ and repeat from Step 1 until a stop criterion is met.

The benefit of using Equation (3) versus (2) is that accumulating over the top-$v\lambda$ best fits to the target allows for explorations of function compositions that contain desired components, but are not fully developed. For example, if we train an implicit function with OccamNet, such as the hyperbola $x_0x_1 = 1$, then the constant function $f = 1$ is always a best fit. However, $f = 1$ does not capture the desired behavior. While a composition that contains x_0 might not be fully developed to x_0x_1, the probability of choosing x_0 should be increased, which is possible through (3). In practice, we find that reweighting the importance of the top-$v\lambda$ routes, substituting $K'_j(\mathcal{M}, g_{i,j}) = K_j(\mathcal{M}, g_{i,j})/i$, improves convergence speed by biasing updates towards the best routes as demonstrated in Section 6.

Two-step training To fit constants, we use activation functions with unspecified constants and combine the training process described in Section 3 with a constant fitting training process. The two-step training process works as follows: We first sample a batch $\mathcal{M}$ and a function batch $(\vec{f}_1, \ldots, \vec{f}_R)$. Next, for each function $\vec{f}_i$, we fit the unspecified constants in $\vec{f}_i$ using gradient descent. Finally, we perform the evolutionary training step on the constant-fitted function batch. To increase training speed, we store each function's fitted constants for reuse. See the SM for more details.

Recurrence OccamNet can also be trained to find recurrence relations. To augment the training algorithm, for each sampled function, we compute its recurrence to a maximum depth D, obtaining a collection of RD functions. Training continues similarly to Section 3 in which we compute the corresponding fitness, select the best $v\lambda$, and update the weights. See the SM for more details.

Regularization To improve implicit function fitting, we implement novel non-differentiable regularization terms which punish trivial solutions, discussed in the SM. We define a modified fitness function to incorporate regularization terms: $K'_i(\mathcal{M}, f) = K_i(\mathcal{M}, f) - s \cdot r[f]$, where $s = |\mathcal{M}|/\sqrt{2\pi\sigma^2}$ is the maximum value of $K_i(\mathcal{M}, f)$, and r represents the regularization term. Because $K_i(\mathcal{M}, f)$ acts as a scaling constant for the probability term, subtracting from it reduces the gradient increase of weights corresponding to undesired functions. If $K'_i(\mathcal{M}, f) < 0$, the weights of these undesired functions decrease. This is similar to reducing the variance of REINFORCE-based gradient estimators.

4 Related Work

Symbolic regression OccamNet was partially inspired by EQL network [11–13], a neural network-based symbolic regression system which successfully finds simple analytic functions. Neural Arithmetic Logic Units (NALU) and related models [26, 27] provide neural inductive bias for arithmetic in neural networks, which in principle can fit some benchmarks in Table 1. NALU updates the weights by backpropagating through the activations, shaping the neural network towards a gating interpretation of the linear layers. However, generalizing those models to a diverse set of function bases might be a formidable task: from our experiments, backpropagation through some activation functions (such as division or sine) makes training considerably harder. In a different computation paradigm, genetic programming (GP) has performed exceptionally well at symbolic regression [28, 29], and a number of evolution-inspired, probability-based models have been explored for this goal [30].

A concurrent work [31] explores deep symbolic regression by using an RNN to search the space of expressions by autoregressive generation of expressions. Interestingly, the authors observed that a risk-aware reinforcement learning approach is a necessary component in their optimization, which is similar to our approach of selecting the top λ function for optimization in Step 2 of our algorithm. A notable difference is that OccamNet does not generate the expressions autoregressively, while it still exhibits gradual increase in modularity during training, as discussed in Section 6. This is also a benefit both for speed and scalability. Moreover, their entropy regularization is an alternative to our regularization. Marrying our approach with theirs is a promising direction for future work.

Program synthesis For programs, one option to fit programs is to use EQL-based models with logic activations (step functions, MIN, MAX, etc.) approximated by sigmoid activations. Another is probabilistic program induction using domain-specific languages [32–34]. Neural Turing Machines [35, 36] and their stable versions [37] are also able to discover interpretable programs, simulated by neural networks via observations of input-output pairs by relying on an external memory. Balog et al.

[38] first train a machine learning model to predict a DSL based on input output pairs and then use a methods from satisfiability modulo theory [39] to search the space of programs built using the predicted DSL. In contrast, our DSL is lower level and can fit components like "sort" instead of including them in the DSL directly. Kurach et al. [40] develop a neural model for simple algorithmic tasks by utilizing memory access for pointer manipulation and dereferencing. However, here we achieve similar results (for example, sorting) without external memory and in only minutes on a CPU.

Integration with deep learning   We are not aware of classifiers that predict MNIST [41][3] or ImageNet [42][4] labels using symbolic rules as considered in the next section. The closest baseline we found is using GP [43], which performs comparably well to our neural method, but cannot easily integrate with deep learning. In the reinforcement learning (RL) domain, Such et al. [7], Salimans et al. [44] propose to train deep models of millions of parameters on standard RL tasks using a gradient-free GP, which is competitive to gradient-based RL algorithms.

SCGs and pruning   Treating the problem of finding the correct function or program as a stochastic computational graph is appealing due to efficient soft approximations to discrete distributions [19–21]. Our T-softmax layers offer such an approximation and could further benefit from an adaptive softmax methodology [45], which we leave for future work. Furthermore, the sparsity induced by T-softmax layers parallels the abundant work on pruning connections and weights in neural networks [46, 47] or using regularizations, encouraging sparse connectivity [48, 14].

5   Experiments

To empirically validate our model, we first develop a diverse collection of benchmarks in five categories: *Analytic functions*, simple, smooth functions; *Implicit functions*, functions specifying an implicit relationship between inputs; *Programs*, non-differentiable operations; *Image/Pattern Recognition*, patterns explained by analytic expressions.

For our experiments, we terminate learning when the top-$v\lambda$ sampled functions all return the same fitness $K(\cdot, f)$ for 30 consecutive epochs. If this happens, these samples are equivalent function expressions. Computing the most likely DAG allows retrieval of the final expression. If this final expression matches the correct function, we determine that the network has converged. For pattern recognition, there is no correct target composition, so we measure the accuracy of the classification rule on a test split, as is conventional.

In all experiments, if termination is not met in a set number of steps we consider it as not converged. We also keep a constant temperature for all the layers except for the last one. An increased last layer temperature allows the network to explore higher function compositionality, as shallow layers can be further trained before the last layer probabilities become concentrated; this is particularly useful for learning functions with high degrees of nesting. More details on hyperparameters for experiments are in the SM. Our network converges rapidly, often in only a few seconds and at most a few minutes.

We first compare the results with that of Eureqa, a software package for symbolic regression [28]. Eureqa uses an evolutionary strategy to fit functions and is highly optimized, allowing it to evaluate billions of functions per second. In the successful cases, Eureqa is able to find the correct answer on the order of 1 second, which is often significantly faster than OccamNet. However, although Eureqa serves as a useful baseline, we emphasize that Eureqa, unlike our architecture, cannot be easily integrated with neural networks. Thus, the benchmark results should not be taken as an exact one-to-one comparison. The results are shown in Table 1 and we discuss them below.

Analytic and implicit functions   We highlight the large success rate for the function $f(x) = (x^2 + x)/(x + 2)$ (line 4 in Table 1), which we originally speculated could easily trick the network with the local minimum $f(x) \approx x - 1$ for large enough x. In contrast, as with the difficulties faced by Udrescu and Tegmark [29], we find that $f(x_0, x_1) = x_0^2(x_0 + 1)/x_1^5$ (line 5) often failed to converge because the factor $x_0^2(x_0 + 1)$ was approximated to x_0^3; even when convergence did occur, it required a relatively large number of steps for the network to resolve this additional constant factor. Notably, Eureqa had difficulty finding the equation $\sum_{n=1}^{3} \sin(nx)$ (line 3). Eureqa was unable to fit

---

[3]Creative Commons Attribution Share Alike 3.0 License
[4]The Creative Commons Attribution (CC BY) License

Table 1: Holistic benchmarking. η is the success rate from 10 trials, the proportion of trials which converge to the correct function. *sec.* is the average number of seconds for convergence. η_b is the baseline success rate (out of 10) run for the allotted seconds in the table. A is the best accuracy from 10 trials, and A_b is the baseline accuracy. For all but "Image Recognition," the baseline is Eureqa. For "Image Recognition," the baseline above the mid-line is HeuristicLab [49] and the baseline below the mid-line is a feed-forward neural network with the same number of parameters as OccamNet.

Analytic Functions

#	Targets	η	sec.	η_b
1	$2x^2 + 3x$	1.0	5	1.0
2	$\sin(3x+2)$	0.8	56	1.0
3	$\sum_{n=1}^{3}\sin(nx)$	0.7	190	0.0
4	$(x^2+x)/(x+2)$	0.9	81	0.7
5	$x_0^2(x_0+1)/x_1^5$	0.3	305	1.0
6	$x_0^2/2 + (x_1+1)^2/2$	0.6	83	0.7
7	$10.5x^{3.1}$	1.0	553	*0.0
8	$\cos(x)$	0.8	410	1.0

Implicit Functions

#	Targets	η	sec.	η_b
9	$x_0 x_1 = 1$	1.0	294	1.0
10	$x_0^2 + x_1^2 = 1$	1.0	153	0.6
11	$x_0/\cos(x_1) = 1$	1.0	131	1.0
12	$x_1/x_0 = 1$	0.9	232	1.0
13	$m_1 v_1 - m_2 v_2 = 0$	1.0	270	0.0

Image Recognition

#	Targets	A	sec.	A_b
22	MNIST Binary	92.9	150	92.8
23	MNIST Trinary	59.6	400	81.2
24	ImageNet Binary	70.7	400	78.0

Programs

#	Targets	η	sec.	η_b
14	$3x$ if $x > 0$, else x	0.7	26	1.0
15	x^2 if $x > 0$, else $-x$	1.0	10	1.0
16	x if $x > 0$, else $\sin(x)$	1.0	236	1.0
17	$\mathsf{SORT}(x_0, x_1, x_2)$	0.7	81	1.0
18	$4\mathsf{LFSR}(x_0, x_1, x_2, x_3)$	1.0	14	1.0
19	$y_0(\vec{x}) = x_1$ if $x_0 < 2$, else $-x_1$; $\ \ y_1(\vec{x}) = x_0$ if $x_1 < 0$, else x_1^2	0.3	157	0.1
20	$g(x) = x^2$ if $x < 2$, else $x/2$; $\ \ y(x) = g^{\circ 4}(x)$	1.0	64	0.0
21	$g(x) = x + 2$ if $x < 2$, else $x - 1$; $\ \ y(x) = g^{\circ 2}(x)$	1.0	64	0.6
25	Backprop OccamNet	98.1	37	97.7
26	Finetune ResNet	97.3	200	95.4

the function $10.5e^{3.1}$, because it cannot fit noninteger exponents. However, Eureqa does find close approximations such as $11.6x^3 + 0.161x^4$, which has an R^2 value of 0.999995. For $\cos(x)$ (line 8 in Table 1) we investigated whether OccamNet could discover a formula for cosine using only the bases $\sin(\cdot)$, $+(\cdot, \cdot)$, and $\div(\cdot, \cdot)$ and the constants 2 and π. We expected OccamNet to discover $\cos(x) = \sin(x + \pi/2)$, but, interestingly, it instead always identified the double angle identity $\cos(x) = \sin(2x)/(2\sin(x))$. We also tested whether OccamNet could discover Taylor polynomials of e^x. OccamNet identified $e^x \approx 1 + x + x^2/2$, but was unable to discover the subsequent $x^3/6$ term. We discuss implicit functions in Section 6, since on them OccamNet demonstrates sizable advantage.

Programming We benchmark the ability to find several non-differentiable, potentially recursive/iterative functions. From our experiments, we highlight both the network's fast convergence to the right functional form and the discovery of the correct recurrence depth of the final expression. This is pronounced in line 20 in Table 1, which is a challenging chaotic series on which Eureqa struggles. We also investigated the usage of bases such as MAX and MIN for the purpose of sorting numbers (line 17), obtaining relatively well-behaved final solutions: the few solutions that did not converge fail only in deciding the second component y_2 of the output vector. Finally, we introduced binary operators and discrete input sets for testing a simple 4-bit LFSR (line 18), the program $(x_0, x_1, x_2, x_3) \rightarrow (x_0 + x_3 \mod 2, x_0, x_1, x_2)$, which converges fast with a high success rate.

Image Recognition We train OccamNet to classify MNIST in a binary setting between the digits 0 and 7 (*MNIST Binary*). For this high-dimensionality task, we implement OccamNet on an Nvidia V100 GPU, yielding sizable 8x speed increase compared to a CPU. In line 22 one of the successful functional fits that OccamNet finds is $y_0(\vec{x}) = \tanh\left(10(\max(x_{715}, x_{747}) + \tanh(x_{435}) + 2x_{710} + 2x_{713})\right)$ and $y_1(\vec{x}) = \tanh\left(10\tanh(10\,(x_{512} + x_{566}))\right)$. The model learns to incorporate pixels into the functional fit that are indicative of the class: here x_{512} and x_{566} are indicative of the digit 7. These observations hold true when we further benchmark the integration of OccamNet with deep feature extractors (lines 24-26). We extract features from ImageNet images using a ResNet 50 model, pre-trained on Imagenet [23]. For simplicity, we select two classes, "minivan" and "porcupine," (*ImageNet Binary*). OccamNet significantly improves its accuracy backpropagating through our model using a standard cross-entropy signal (lines 25-26). We either freeze the ResNet weights

(*Backprop OccamNet*) or finetune ResNet through OccamNet (*Finetune ResNet*). In both cases the converged OccamNet represents simple rules ($y_0(\vec{x}) = x_{1838}$, $y_1(\vec{x}) = x_{1557}$) suggesting that replacing the head in deep neural networks with OccamNet might be promising.

Real world regression datasets   We also test OccamNet's ability to fit real world datasets, selecting 15 datasets with 1666 or fewer datapoints from the Penn Machine Learning Benchmarks (PMLB[5]) regression datasets [50]. We compare OccamNet to a genetic algorithm implemented using DEAP [51] with Epsilon-Lexicase (Eplex) selection [52], which was identified in a large benchmark study by Orzechowski et al. [53] as the top-performing genetic method for modeling data, in terms of validation loss for PMLB tabular regression datasets. We also compare OccamNet to AI Feynman 2.0 (AIF) [29, 54], a state of the art method in symbolic regression.[6]

We test OccamNet twice. For the first test, which we label "OccamNet," we test exactly 1,000,000 functions, the same number as we test for Eplex. For the second test, which we label "V100," we exploit our architecture's integration with the deep learning framework by running OccamNet on an Nvidia V100 GPU and testing a much larger number of functions. We allow AIF to run for approximately as long or longer than OccamNet for each dataset. We perform grid search on hyperparamenters, discussed in the SM, and identify the fits with the best training, validation, and testing Mean Squared Error (MSE) losses. The raw data from these experiments is shown in the SM.

Figure 3 shows the relative performance of OccamNet and comparison datasets according to several metrics. As shown in Figure 3a,b,c, overall Eplex outperforms OccamNet in training and testing MSE loss, but OccamNet outperforms Eplex in validation loss. We speculate that OccamNet's performance drop between the validation and testing datasets results from overfitting from the larger set of hyperparameter combinations for OccamNet (details in the SM).

Additionally, OccamNet runs faster than Eplex in nearly all datasets tested, often by an order of magnitude (Figure 3d). Furthermore, OccamNet is highly parallel and can easily scale on a GPU. Thus, a major advantage of OccamNet is its speed and scalability. Comparing V100 and Eplex demonstrates that OccamNet continues to improve when testing more functions. V100 performs worst in the comparison based on testing MSE (see Figure 3e), but it still outperforms Eplex at 10/15 of the datasets, while running more than 9 times faster. Thus OccamNet's speed and scalability can be exploited to greatly increase its accuracy at symbolic regression. This demonstrates that OccamNet is a powerful alternative to genetic algorithms for interpretable data modeling.

OccamNet also outperforms AIF for training, validation, and testing, while running faster. OccamNet acheives a lower training and validation MSE than AIF for every dataset tested. For training loss, OccamNet performs better than AIF in 4/7 datasets (Figure 3f). V100 performs even better. Additionally, AIF performs polynomial fitting, which OccamNet does not, giving it an additional advantage. However, the datasets we test are a worst case for AIF; the datasets are small, have no known underlying formula, and we normalize the data prior to training, meaning that AIF will likely struggle not to overfit with its neural network and will also be less likely to find graph modularities.

6   Discussion

Limitations   Since our experimental settings did not require very large depths, we have not tested the limits of OccamNet in terms of depth rigorously (preliminary results on increasing the depth for pattern recognition are in the SM). We expect increasing depth to yield significant complications, as the search space would become exponentially large. We recognize the need of creating symbolic regression benchmarks that would require expressions that are large in depth. We believe that concurrent contributions [31] would also benefit from such benchmarks. Another direction where OccamNet might be improved is low level optimization that would make the method more efficient to train. For example, in our PMLB experiments, we estimate that OccamNet performs >8x as many computations as necessary. Eplex may also benefit from optimization. Finally, similarly to other symbolic regression methods, OccamNet requires a specified basis to fit a dataset. While it is a

---

[5]Creative Commons Attribution 4.0 International License

[6]AIF's regression algorithm examines all possible feature subsets, which grows exponentially with the number of features. Accordingly, we only test the datasets with 10 or fewer features. AI Feyman failed to run on a few datasets. All remaining datasets are included in tables and figures.
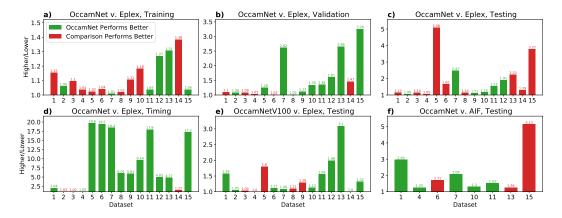
Figure 3: A bar chart showing the relative performance between OccamNet and two baseline methods, Eplex and AIF. The x-axis is the dataset involved. The y axis is the relative performance according to the given metric: the MSE on the training, validation, or testing set, or the training time. To compute this relative performance, we divide the higher (worse) performance value by the lower (better) performance value for each dataset. The green bars represent datasets where OccamNet has a lower (better) performance value than the comparison baseline method, and the red bars represent the datasets where the comparison method has a better performance than OccamNet.

notable advantage of OccamNet to have non-differentiable bases, further work needs to be done to explore optimization at a meta level that discovers appropriated bases for the datasets of interest.

Advantages OccamNet's learning procedure allows it to combine partial solutions into better results. For example in Figure 4, the correct function's probability increases by more than 100 times *before being sampled* because OccamNet samples similar approximate solutions.
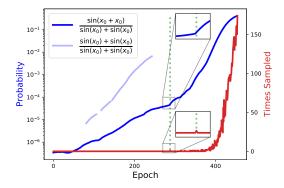

Figure 4: Gradual modularity with training. Dark blue is the correct function. Light blue is a suboptimal fit with high probability early in training. Red corresponds to the correct function. The insets show the first sample of the correct function.

OccamNet successfully fits many implicit functions that other neurosymbolic architectures struggle to fit because of the non-differentiable regularization terms required to avoid trivial solutions. Although Eureqa also fits many of these equations, we find that it sometimes requires the data to be ordered by some latent variable and struggles when the dataset is very small. This is likely because Eureqa numerically evaluates implicit derivatives from the dataset [55], which can be noisy when the data is sparse. While Schmidt and Lipson [55] proposes methods for analyzing unordered data, it is unclear whether these methods have been implemented in Eureqa. Thus, OccamNet seems to shine in its ability to fit unordered and small datasets described by implicit equations (e.g. momentum conservation in line 5 in Table 1).

To our knowledge, a unique advantage of our method is that OccamNet represents complete analytic expressions with a single forward pass, which allows sizable gains when using an AI accelerator, as demonstrated by our experiments on V100 (Figure 3). Furthermore, because of this property, OccamNet, can be easily integrated with components from the standard deep learning toolkit. For example, lines 25-26 in Table 1 demonstrate integration and joint optimization with neural networks, which is not possible with Eureqa. We also conjecture that such integration with autoregressive approaches [31] might be challenging as the memory and latency would increase.

9

7   Conclusion and Future Work

We motivated, introduced and tested OccamNet, a fast neural model for symbolic regression, on a wide range of benchmarks. Potential stakeholders that might benefit from our contributions are institutions and scientists who would like to explain their datasets with symbolic expressions. With OccamNet they might be able to explain their data with analytic expressions quickly and cheaply.

Currently our method is not explicitly designed against adversarial attacks. Thus, malicious stakeholders could exploit our method and manipulate the symbolic fits that OccamNet produces. A potential direction towards alleviating the problem would be to explore ways to robustify OccamNet by training it against an adversary. Therefore, we leave tackling adversarial robustness of neural models for symbolic regression as an exciting direction for future work.

Supplemental Material

We have organized the Supplemental Material as follows:

- In Section A we present further details about our mathematical formalism.

- In Section B we present our initialization algorithms.

- In Section C we present our function selection algorithm.

- In Section 3 we present our method for allowing OccamNet to model recurrence.

- In Section E we discuss our regularization terms.

- In Section F we discuss our methods for accounting for undefined functions.

- In Section G we present our experimental hyperparameters.

- In Section H we present further details about the setup and hyperparameters for our experiments with the PMLB Datasets. We also present our raw results.

- In Section I we examine the models each method provides for the PMLB Datasets.

- In Section J we present a series of ablation studies.

- In Section K we discuss neural models for sorting and pattern recognition.

- In Section L we discuss the evolutionary strategies for fitting functions and programs that we use as benchmarks.

- In Section M, we catalogue our code and video files.

A Mathematical Formalism

Here we introduce the full mathematical formalism behind OccamNet. As described in the main text, we start from a predefined collection of N basis functions $\boldsymbol{\Phi} = \{\phi_i(\cdot)\}_{i=1}^N$. Each neural network layer is defined by two sublayers, the *arguments* and *image* sublayers. For a network of depth L, each of these sublayers is reproduced L times. Now let us introduce their corresponding hidden states: each l-th arguments sublayer defines a hidden state vector $\widetilde{\mathbf{h}}^{(l)}$, and similarly each l-th image sublayer defines a hidden state $\mathbf{h}^{(l)}$, as follows

$$\widetilde{\mathbf{h}}^{(l)} = \left[\widetilde{h}_1^{(l)}, \ldots, \widetilde{h}_M^{(l)}\right], \quad \mathbf{h}^{(l)} = \left[h_1^{(l)}, \ldots, h_N^{(l)}\right]. \tag{4}$$

These vectors are related through the bases functions:

$$h_i^{(l)} = \phi_i\left(\widetilde{h}_{j+1}^{(l)}, \ldots, \widetilde{h}_{j+\alpha(\phi_i)}^{(l)}\right), \quad j = \sum_{0 \leq k < i} \alpha(\phi_k), \quad M = \sum_{0 \leq k \leq N} \alpha(\phi_k), \tag{5}$$

where $\alpha(\phi)$ is the arity of function $\phi(\cdot, \ldots, \cdot)$. This formally expresses how the arguments connect to the images in any given layer, visualized as the bold edges between sublayers in Figure 1 in the main paper. To complete the architecture and connect the images from layer l to the arguments of layer $(l+1)$, we use the described softmax transformation: [7]

$$\mathbf{W}(T) \cdot \mathbf{h}^{(l)} = \begin{bmatrix} \mathsf{softmax}(\mathbf{w}_1; T)^\top \\ \vdots \\ \mathsf{softmax}(\mathbf{w}_{M_{l+1}}; T)^\top \end{bmatrix} \begin{bmatrix} h_1^{(l)} \\ \vdots \\ h_{N_l}^{(l)} \end{bmatrix}$$

$$\equiv \begin{bmatrix} \widetilde{h}_1^{(l+1)} \\ \vdots \\ \widetilde{h}_{M_{l+1}}^{(l+1)} \end{bmatrix} = \widetilde{\mathbf{h}}^{(l+1)}, \tag{6}$$

where the hidden states $\mathbf{h}^{(l)}$ and $\widetilde{\mathbf{h}}^{(l+1)}$ have N_l and M_{l+1} coordinates, respectively.

From Equation (5), we see that $M_{l+1} = M = \sum_{0 \leq k \leq N} \alpha(\phi_k)$. If no skip connections are used, $N_l = N = |\boldsymbol{\Phi}|$. If skip connections are used, however, N_l grows as l increases. We demonstrate how the scaling grows as follows.

Let u be the number of inputs and v be the number of outputs. When learning connections from images to arguments at layer l ($1 \leq l \leq L$) there will be skip connections from the images of the previous $l - 1$ layers $1, \ldots, l - 1$. Hence each layer produces the following list of numbers of images $\{u + (i+1)N\}_{i=0}^L$. We learn linear layers from these images to arguments, and the number of arguments is always M. Thus, in total we have the following number of parameters:

$$v(u + (L+1)N) + M \sum_{i=0}^{L-1} (u + (i+1)N) \in O(NML^2),$$

Along with the added inputs and constants, this description fully specifies the mathematical structure of our architecture.

B Initialization

We originally initialized all model weights to 1. However, this initializes complex functions, which have DAGs with many more edges than simple functions, to low probabilities. As a result, we found in practice that the network sometimes struggled to converge to complex functions with high fitness $K(\mathcal{M}, f)$ because their initial low probabilities meant that they were sampled far less often than simple functions. This is because even if complex functions have a higher probability increase than

---

[7]as before, we define for any $\mathbf{z} = [z_1, \ldots, z_{N_l}]$ the softmax function as follows $\mathsf{softmax}(\mathbf{z}; T) :=$ $\left[\frac{\exp(z_1/T)}{\sum_{i=1}^{N_l} \exp(z_i/T)}, \ldots, \frac{\exp(z_{N_l}/T)}{\sum_{i=1}^{N_l} \exp(z_i/T)}\right]^\top$

simple functions when they are sampled, the initial low probabilities caused the complex functions to be sampled far less and to have an overall lower expected probability increase.

To address this issue, we use a second initialization algorithm, which initializes all functions to equal probability.

This initialization algorithm iterates through the layers of the network. It establishes as an invariant that, after assigning the weights up to the lth layer, all paths leading to a given node in the lth argument layer have equal probabilities. Then, each argument layer node has a unique corresponding probability, the probability of all paths up to that node. We denote the probability of the ith node in the lth argument sublayer as $\widetilde{p}_i^{(l)}$. Because each argument layer node has a corresponding probability, each image layer node must also have a unique corresponding probability, which, for the ith node in the lth image sublayer, we denote as $p_i^{(l)}$. These image layer probabilities are given by

$$p_i^{(l)} = \prod_{k=n+1}^{n+\alpha(\phi_i)} \widetilde{p}_k^{(l)}, \quad n = \sum_{j=1}^{i-1} \alpha(\phi_j). \tag{7}$$

Our algorithm initializes the input image layer's nodes to probability 1. As the algorithm iterates through all subsequent T-Softmax layers, the invariant established above provides a system of linear equations involving the desired connection probabilities, which the algorithm solves. The algorithm groups the previous image layer according to the node probabilities, obtaining a set of ordered pairs $\{(p'^{(l)}_i, n^{(l)}_i)\}_{i=1}^{k}$ representing $n^{(l)}_i$ nodes with probability $p'^{(l)}_i$ in the lth layer. Note that if two image nodes have the same probability, for each P-node in the arguments sublayer, the edges between the image nodes and the P-node must have the same probability in order to satisfy the algorithm's invariant. Then, we define $p'^{(l,j)}_i$ as the probability of the edges between the image nodes with probability $p'^{(l)}_i$ and the jth argument P-node of the lth layer. The probabilities of the edges to a given P-node sum to 1, so for each j, we must have $\sum_i n_i p'^{(l,j)}_i = 1$. Further, the algorithm requires that the probability of a path to a P-node through a given connection is the same as the probability of a path to that P-node through any other connection. The probability of a path to the jth P-node through a connection with probability $p'^{(l,j)}_i$ is $p'^{(l)}_i p'^{(l,j)}_i$, so we obtain the equations $p'^{(l)}_0 p'^{(l,j)}_0 = p'^{(l)}_i p'^{(l,j)}_i$, for all i and j. These two constraints give the vector equation

$$\begin{bmatrix} n_0^{(l)} & n_1^{(l)} & n_2^{(l)} & \cdots & n_k^{(l)} \\ p'^{(l)}_0 & -p'^{(l)}_1 & 0 & \cdots & 0 \\ p'^{(l)}_0 & 0 & -p'^{(l)}_2 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ p'^{(l)}_0 & 0 & 0 & \cdots & -p'^{(l)}_k \end{bmatrix} \begin{bmatrix} p'^{(l,j)}_0 \\ p'^{(l,j)}_1 \\ p'^{(l,j)}_2 \\ \vdots \\ p'^{(l,j)}_k \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix},$$

for all $1 \le j \le M$. The algorithm then solves for each $p'^{(l,j)}_i$.

After determining the desired probability of each connection of the lth layer, the algorithm computes the SPL weights $\mathbf{w}'^{(l,j)}$ that produce the probabilities $p'^{(l,j)}_i$. Since there are infinitely many possible weights that produce the correct probabilities, the algorithm sets $w'^{(l,j)}_0 = 0$. Then, the algorithm uses the softmax definition of the edge probabilities to determine the required value of $\sum_{m=1}^{k} \exp\left(w'^{(l,j)}_i / T^{(l)}\right)$:

$$p'^{(l,j)}_0 = \frac{\exp\left(w'^{(l,j)}_0 / T^{(l)}\right)}{\sum_{m=1}^{k} \exp\left(w'^{(l,j)}_m / T^{(l)}\right)}$$

$$= \frac{1}{\sum_{m=1}^{k} \exp\left(w'^{(l,j)}_m / T^{(l)}\right)}$$

so

$$\sum_{m=1}^{k} \exp\left(w'^{(l,j)}_m / T\right) = 1/p'^{(l,j)}_0.$$

Substituting this equation into the expression for the other probabilities gives

$$p'^{(l,j)}_i = \exp\left(w'^{(l,j)}_i/T^{(l)}\right) / \left(\sum_{m=1}^{k} \exp\left(w_i/T^{(l)}\right)\right)$$

$$= p'^{(l,j)}_0 \exp\left(w'^{(l,j)}_i/T^{(l)}\right).$$

Solving for $w'^{(l,j)}_i$ gives

$$w'^{(l,j)}_i = T^{(l)} \log\left(p'^{(l,j)}_i / p'^{(l,j)}_0\right),$$

which the algorithm uses to compute $w'^{(l,j)}_i$.

After determining the weights $w'^{(l,j)}_i$ the algorithm assigns them to the corresponding $w^{(l,j)}_i$. In particular, if the ith image node has probability $p'^{(l)}_k$, the weights of edges to the ith node are given by $w^{(l,j)}_i = w'^{(l,j)}_k$, for all j. The algorithm then determines the values of $\widehat{p}^{(l+1)}_i$, given by $\widehat{p}^{(l+1)}_i = p^{(l)}_1 p^{(l,i)}_1$. Finally, the algorithm determines $p^{(l+1)}_i$ using Equation 7 and repeats the above process for subsequent layers until it reaches the end of the network.

This algorithm efficiently equalizes the probabilities of all functions in the network. In practice, however, we find that perfect equalization of functions causes activation functions with two inputs to be highly explored. This is because there are many more possible functions containing activation functions with two inputs than with one input. In practice, therefore, we find that a balance between initializing all weights to one and initializing all functions to equal probability is most effective for exploring all types of functions.

To implement this balance, we create an equalization hyperparameter, E. If $E = 0$, we initialize all weights to 1 as in the original OccamNet architecture. If $E \neq 1$, we use the algorithm presented above to initialize the weights, and then divide all of the weights by E. For $E > 1$, this has the effect of initializing weights between the two initialization approaches. In practice, we find that values of $E = 1$ and $E = 5$ are most effective for exploring all types of functions (See Section H).

C Function Selection

As discussed in the main text, after training using an evolutionary strategy, the network selects the function $\vec{f}$ with the highest probability $q(\vec{f}|\mathbf{W})$.

We develop a dynamic programming algorithm which determines the DAG with the highest probability. The algorithm steps sequentially through each argument layer, and at each argument layer it determines the maximum probability path to each argument node. Knowing the maximum probability paths to the previous argument layer nodes allows the algorithm to easily determine the maximum probability paths to the next argument layer.

As with the network initialization algorithm, the function selection algorithm associates the ith P-node of the lth argument sublayer with a probability, $\widehat{p}^{(l)}_i$, which represents the highest probability path to that node. Similarly, we let $p^{(l)}_i$ represent the assigned probability of the ith node of the lth image sublayer, defined as the highest probability path to a given image node. $p^{(l)}_i$ can once again be determined from $\widehat{p}^{(l)}_i$ using Equation 7. Further, the algorithm associates each node with a function, $\widetilde{f}^{(l)}_i$ for argument nodes and $f^{(l)}_i$ for image nodes, which represents the highest probability function to the corresponding node. Thus, $\widetilde{f}^{(l)}_i$ has probability $\widehat{p}^{(l)}_i$, and $f^{(l)}_i$ has probability $p^{(l)}_i$. Further, $f^{(l)}_i$ is determined from $\widetilde{f}^{(l)}_i$ using

$$f^{(l)}_i(\vec{x}) = \phi_i\left(\widetilde{f}^{(l)}_{n+1}(\vec{x}), \dots, \widetilde{f}^{(l)}_{n+\alpha(\phi_j)}(\vec{x})\right), \quad n = \sum_{j=1}^{i-1} \alpha(\phi_j). \tag{8}$$

The algorithm iterates the the networks layers. At the lth layer, it determines the maximum probability path to each argument node, computing

$$\widetilde{p}_i^{(l+1)} = \text{MAX}\left(p_0^{(l)} p_0^{(l,i)}, \dots, p_N^{(l)} p_N^{(l,i)} \right)$$

$$\widetilde{f}_i^{(l+1)} = \begin{cases} f_0^{(l)} & \text{if } \widetilde{p}_i^{(l+1)} = p_0^{(l)} p_0^{(l,i)} \\ f_1^{(l)} & \text{if } \widetilde{p}_i^{(l+1)} = p_1^{(l)} p_1^{(l,i)} \\ \vdots & \vdots \\ f_N^{(l)} & \text{if } \widetilde{p}_i^{(l+1)} = p_N^{(l)} p_N^{(l,i)} \end{cases}.$$

Next, it determines the maximum probability path up to each image node, computing $p_i^{(l+1)}$ and $f_i^{(l+1)}$ using Equations 7 and 8, respectively. The algorithm repeats this process until it reaches the output layer, at which point it returns $\vec{f}_{\max} = [\widetilde{f}_1^{(L)}, \dots, \widetilde{f}_N^{(L)}]^\top$ and $p_{\max} = \prod_{i=1}^{N} \widetilde{p}_i^{(L)}$.

D Recurrence

OccamNet can also be trained to find recurrence relations, which is of particular interest for programs that rely on FOR or WHILE loops. To find such recurrence relations, we assume a *maximal* recursion D. We use the following notation for recurring functions: $f^{\circ(n+1)}(x) \equiv f^{\circ n}(f(x))$, with base case $f^{\circ 1}(x) \equiv f(x)$.

To augment the training algorithm, we first sample $(\vec{f}_1, \dots, \vec{f}_R) \sim q(\cdot | \mathbf{W}^{(t)})$. For each $\vec{f}_i$, we compute its recurrence to depth D as follows $\left(\vec{f}_i^{\circ 1}, \vec{f}_i^{\circ 2}, \dots, \vec{f}_i^{\circ D} \right)$, obtaining a collection of RD functions. Training then continues as usual; we compute the corresponding $K_j(\mathcal{M}, \vec{f}_{i(j)}^{\circ n})$, select the best $v\lambda$, and update the weights. It is important to note that we consider all depths up to D since our maximal recurrence depth might be larger than the one for the target function.

Note that we do not change the network architecture to accommodate for recurrence depth $D > 1$. As described in the main text, we can efficiently use the network architecture to evaluate a sampled function $\vec{f}(\vec{x})$ for a given batch of $\vec{x}$. To incorporate recurrence, we take the output of this forward pass and feed it again to the network D times, similar to a recurrent neural network. The resulting outputs are evaluations $\left(\vec{f}_i^{\circ 1}(\vec{x}), \vec{f}_i^{\circ 2}(\vec{x}), \dots, \vec{f}_i^{\circ D}(\vec{x}) \right)$ for a given batch of $\vec{x}$.

E Regularization

As discussed in the main text, to improve implicit function fitting, we implement a regularized loss function,

$$K_i'(\mathcal{M}, f) = K_i(\mathcal{M}, f) - s \cdot r[f],$$

for some regularization function r, where $s = n(\mathcal{M})/\sqrt{2\pi\sigma^2}$ is the maximum possible value of $K_i(\mathcal{M}, f)$. We define

$$r[f] = w_\phi \cdot \phi[f] + w_\psi \cdot \psi[f] + w_\xi \cdot \xi[f] + w_\gamma \cdot \gamma[f],$$

where $\phi[f]$ measures trivial operations, $\psi[f]$ measures trivial approximations, $\xi[f]$ measures the number of constants in f, $\gamma[f]$ measures the number of activation functions in f, and w_ϕ, w_ψ, w_ξ, and w_γ are weights for their respective regularization terms. We now discuss each of these regularization terms in more detail.

A The Phi Term

The $\phi[f]$ term measures whether the unsimplified form of f contains trivial operations, operations which cause an expression to simplify. For example, division is a trivial operation in x/x, because the expression simplifies to 1. Similarly, $1 \cdot x$, x^1, and x^0 are all trivial operations. We punish these trivial operations because they often produce constant outputs without ever adding meaning to an expression.

To detect trivial operations, we employ two procedures. The first uses the `SymPy` package [56] to simplify f. If the simplified expression is different from the original expression, then there are trivial operations in f and this procedure returns 1. Otherwise the first procedure returns 0. Unfortunately, the `SymPy` == function to test if functions are equal often incorrectly indicates that nontrivial functions are trivial. For example, `SymPy`'s `simplify` function, which we use to test if a function can be simplified, converts $x + x$ to $2 \cdot x$, and the == function states that $x + x \neq 2 \cdot x$. To combat this, we develop a new function, `sympyEquals` which corrects for these issues with ==. The `sympyEquals` is equivalent to ==, except that it does not take the order of terms into account, and it does not mark expressions such as $x + x$ and $x \cdot x$ as unsimplified. We find that this greatly improves function fitting.

The constant fitting procedure often produces functions which only differ from a trivial operation because of imperfect constant fitting, such as $f(x_0) = x_0^{0.0001}$, which is likely meant to represent x_0^0. SymPy, however, will not mark this function as trivial. The second procedure addresses this issue by counting the constant activations, such as $x_0^{0.0001}$, $1.001 \cdot x_0$, and $x_0 + 0.001$, which approximate trivial operations. For the activation function $f(x) = x + c$, if the fitted c satisfies $-0.1 < c < 0.1$, the procedure adds 1 to its counter. Similarly, for the activation functions $f(x) = cx$ and $f(x) = x^c$, if the fitted c satisfies $-0.1 < c < 0.1$ or $0.9 < c < 1.1$, the procedure adds 1 to its counter. We select these ranges to capture instances of imperfect constant fitting without labeling legitimate solutions as trivial. After checking all activation functions used, the procedure returns the counter.

The $\phi[f]$ term returns the sum of the outputs of the first and second procedures. We find that a weight of $w_\phi \approx 0.7$ for $\phi[f]$ is most effective in our loss function. This value of w_ϕ ensures that most trivial f have $K_i(\mathcal{M}, f) - s \cdot w_\phi \cdot \phi[f] < 0$, thus actively reducing the weights corresponding to functions with trivial operations, without overpunishing functions and hindering learning.

B The Psi Term

When punishing trivial operations using the ϕ term, we find that the network discovers many nontrivial operations which very closely approximate trivial functions by exploiting portions of functions with near-zero derivatives, which can be used to artificially compress data. For example, $\cos(x/2)$ closely approximates 1 if $-1 < x < 1$. Unfortunately, it is often difficult to determine if a function approximates a trivial function simply from its symbolic representation. This issue is also identified in [55].

To detect these trivial function approximations, we develop an approach which analyzes the activation functions' outputs *during* the forward pass. The $\psi[f]$ term counts the number and severity of activation functions which, during a forward pass, the network identifies as possibly approximating trivial solutions. For each basis function, the network stores values around which outputs of that function often cluster artificially. Table 2 lists the bases which the network tests for clustering.

The procedure for determining ψ is as follows. The algorithm begins with a counter of 0. During the forward pass, the if the network reaches a basis function ϕ listed in Table 2, the algorithm tests each ordered tuple (ϕ, a, δ) from Table 2, where a is the point tested for clustering and δ is the clustering tolerance. If the mean of all the outputs of the basis function, $\bar{y}$, for a given batch satisfies $|\bar{y} - a| < \delta$, the algorithm adds $\min(5, 0.1/|\bar{y} - a|)$ to the counter. These expressions increase with the severity of clustered data; the more closely the outputs are clustered, the higher the punishment term. The minimum term ensures that $\psi[f]$ is never infinite.

We also test for the approximation $\sin(x) \approx x$, by testing the inputs and outputs of the sine basis function. If the inputs and outputs x and y to the sine basis satisfy $\overline{|y - x|} < 0.1$, the algorithm adds $\min(5, 0.05/\overline{|y - x|})$ to the counter. In the future, we plan to consider more approximations similar to the small angle approximation.

$\psi[f]$ should not artificially punish functions involving the bases listed in Table 2 that are not trivial approximations, because no proper use of these basis functions will always produce outputs very close to the clustering points. Because $\psi[f]$ flags functions based on their batch outputs, each batch will likely give different outcomes. This allows $\psi[f]$ to better discriminate between trivial function approximations and nontrivial operations: $\psi[f]$ should flag trivial function approximations often, but it should only flag nontrivial operations rarely, when the inputs statistically fluctuate to produce clustered outputs. In practice, we find that a weight of $w_\psi \approx 0.3$ for $\psi[f]$ is most effective in our loss function.

Table 2: Basis functions tested for clustering

Basis Functions	Cluster Points	Cluster Tolerance
$(\cdot)^2$	$\{0\}$	0.25
$(\cdot)^3$	$\{0\}$	0.25
$\sin(\cdot)$	$\{1, -1\}$	0.25
$\cos(\cdot)$	$\{1, -1\}$	0.25
$(\cdot)^c$	$\{1\}$	0.5

C   The Xi Term

When our network converges to the correct solution, it may converge to a more complicated expression equivalent to the desired expression. To promote simpler expressions, we slightly punish functions based on their complexity. The $\xi[f]$ term counts the number of activation functions used to produce f, which serves as a measure of f's complexity. We find that a small weight of $w_\xi \approx 0.1$ for $\xi[f]$ is most effective in our loss function. This small value has little significance when distinguishing between a function which fits a dataset well and a function which does not, but it is enough to promote simpler functions over complex functions when they are otherwise equivalent.

D   The Gamma Term

The $\gamma[f]$ term also punishes functions for their complexity. The $\gamma[f]$ term counts the number of constants in f, which, like the number of activation functions, serves as a metric for f's complexity. We find that a weight of $w_\gamma \approx 0.15$ for $\gamma[f]$ is most effective in our loss function. Just as with $\xi[f]$, this small value has little significance when distinguishing between a function which fits a dataset well and a function which does not, but it is enough to slightly promote simpler functions over complex functions when they are otherwise equivalent. We weight $\gamma[f]$ slightly higher than $\xi[f]$ because many functions with constants can be simplified.

F   Functions with Undefined Outputs

One difficulty that may arise when training OccamNet is that many sampled functions are undefined on the input data range. Two cases of undefined functions are: 1) the function is undefined on part of the input data range for all values of a set of constants or 2) the function is only undefined when the function's constants take on certain values. An example function satisfying case 1 is $f_1(x_0) = c_0/(x_0 - x_0)$, which divides by 0 regardless of the value of c_0. An example function satisfying case 2 is $f_2(x_0) = x_0^{c_0}$, which is undefined whenever x_0 is negative and c_0 is not an integer.

In the first case, the network should abandon the function. In the second case, the network should try other values for the constants. However, the network cannot easily determine which case an undefined function satisfies. To balance both cases, if the network obtains an undefined result, such as `NaN` or `inf`, for the forward pass, the network tests up to 100 more randomized sets of constants. If none of these attempts produce defined results, the network returns the array of undefined outputs. For example, with $c_0/(x_0 - x_0)$, the network tests a first set of constants, determines that they produce an undefined output, and tests 100 more constants. None of these functions are defined on all inputs, so the network returns the undefined outputs.

In contrast, with $x_0^{c_0}$, the network might find that the first set of constants produces undefined outputs, but after 20 retries, the network might discover that $c_0 = 2$ produces a function defined on all inputs. The network will then perform gradient descent, and return the fitted value of c_0. Further, if at any point in the gradient descent, the forward pass yields undefined results, the network returns the well-defined constants and associated output from the previous forward pass. For example, for $x_0^{c_0}$, after the network discovers that $c_0 = 2$ works, the gradient for the constants will be undefined because c_0 can only be an integer. Thus, the network will return the outputs of $x_0^{c_0}$, for $c_0 = 2$, before the undefined gradients.

We find that if the network simply ignores functions with undefined outputs, these functions will increase in probability, because our network regularization punishes many other functions. Since these punished functions decrease in probability during training, the functions with undefined outputs

begin to increase in probability. To combat this, instead of ignoring undefined functions, we use a modified fitness for undefined functions, $K_i'(\mathcal{M}, f) = -w_s s$, where $s = n(\mathcal{M})/\sqrt{2\pi\sigma^2}$ is the maximum possible value of $K_i(\mathcal{M}, f)$ and w_s is a hyperparameter than can be tuned. This punishes undefined functions, causing their weights to decrease. In practice, we find that a value of w_s between 0 and 1 is most effective, depending on the application.

G   Experimental Hyperparameters

In Tables 3 and 4, we present and detail the hyperparameters we used for our experiments in the main paper. We present our experimental setup for our experiments with PMLB Datasets separately, in Section H. Note that detail about the setup for each experiment is provided in the attached code.

In Tables 3 and 4, $+$ is addition (2 arguments); $-$ is subtraction (2 arguments) $\cdot$ is multiplication (2 arguments); $/$ is division (2 arguments); $\sin(\cdot)$ is sine, $+c$ is addition of a constant, $\cdot c$ is multiplication of a constant, $(\cdot)^c$ is raising to the power of a constant, $\leq$ is a an if-statement (4 arguments: comparing two numbers, one return for a true statement, and one for a false statement); $-(\cdot)$ is negation. MIN, MAX and XOR all have 2 arguments. Here, SIGMOID$'$ is a sigmoid layer and $\tanh'$ is a tanh layer where the inputs to both functions are scaled by a factor of 10, $+_4$ and $+_9$ are the operations of adding 4 and 9 numbers respectively, and MAX$_4$, MIN$_4$, MAX$_9$ and MIN$_9$ are defined likewise. The bases for pattern recognition experiments are given as follows: $\mathbf{\Phi}_A$ consists of SIGMOID$'$, SIGMOID$'$, $\tanh'$, $\tanh'$, $+_4$, $+_4$, $+_9$, $+_9$, $+$, $+$, MIN, MIN, MAX and MAX; $\mathbf{\Phi}_B$ consists of id, id, id, id, $+$, $+$, $+$, $+_4$, $+_4$, $+_9$, $+_9$, $+_9$, $\tanh,$, $\tanh$, SIGMOID, and SIGMOID. Additionally, the constants used for pattern recognition are $\mathbf{C} = \{-1, -1, 0, 0, 1, 1, 1\}$.

In Tables 3 and 4, L is the depth, T is the temperature, T_{last} is the temperature of the final layer, σ is the variance, R is the sample size, λ is the fraction of best fits, α is the learning rate, E is the initialization parameter described in Section B, and w_ϕ, w_ψ, w_ξ, and w_γ are as defined in Appendix E. Table 3 does not include E as a listed hyperparameter, because for all experiments listed $E = 0$. With * we denote the experiments for which the best model is without skip connections. We do not regularize for any experiments in Table 3. NA entries mean that the correspodning hyperparameter is not present in the experiment.

For all experiments in Table 4, we use a learning rate of 0.01, and, when applicable, a constant-learning rate of 0.05. We also set the temperature to 1 and the final layer temperature to 10 for all experiments in the table. For the equation $m_1 v_1 - m_2 v_2 = 0$, we sample m_1, v_1, and m_2 from $[-10, 10]$ and compute v_2 using the implicit function.

All our experiments in Table 3 use a batch size of 1000, except for *Backprop OccamNet* and *Finetune ResNet*, for which we use batch size 128. All our experiments in Table 4 use a batch size of 200. For each of our pattern recognition experiments we use a 90%/10% train/test random split for the corresponding datasets. The input pixels are normalized to be in the range [0, 1]. During validation: for *MNIST Binary*, *MNIST Trinary* and *ImageNet Binary* the outputs of OccamNet are thresholded at 0.5 and if the output matches the one-hot label, then the prediction is accurate and it is inaccurate otherwise; for *Backprop OccamNet* and *Finetune ResNet* the outputs of OccamNet are viewed as the logits of a negative log likelihood loss function, so the prediction is the argmax of the logits. Backprop OccamNet and Finetune ResNet use an exponential decay of the learning rate with decay factor 0.999.

H   PMLB Experiment Setup and Results

As described in the main text, we test OccamNet on 15 datasets from the Penn Machine Learning Benchmarks (PMLB) repository [50]. The 15 datasets chosen, and the corresponding numbers we use to reference them, are shown in Table 5. We chose these datasets by selecting the first 15 regression datasets with fewer than 1667 datapoints. These 15 datasets are the only datasets from PMLB we examine.

We test four methods on these datasets. OccamNet, V100, Eplex, AIF, and Extreme Gradient Boosting (XGB) [57]. We have described all of these methods except for XGB in the main text. XGB is a tree based method which was identified by [53] as the best machine learning method based on validation MSE for modeling the PMLB datasets. However, XGB is not interpretable and thus

Table 3: Hyperparameters for Experiments Where $E = 0$

Target	Bases	Constants	Range	$L/T/T_{\text{last}}/\sigma$	$R/\lambda/\alpha$
		Analytic Functions			
$2x^2 + 3x$	$\langle\cdot,\cdot,+,+\rangle$	$\emptyset$	$[-10, 10]$	2/1/1/0.01	50/5/0.05
$\sin(3x + 2)$	$\langle\cdot,\sin,\sin,+,+\rangle$	1, 2	$[-10, 10]$	3/1/1/0.001	50/5/0.005
$\sum_{n=1}^{3}\sin(nx)$	$\langle\sin,\sin,+,+,+\rangle$	1, 2	$[-20, 20]$	5/1/1/0.001	50/5/0.005
$(x^2 + x)/(x + 2)$	$\langle\cdot,\cdot,+,+,/,/\rangle$	1	$[-6, 6]$	2/1/2/0.0001	100/5/0.005
$x_0^2(x_0 + 1)/x_1^5$	$\langle\cdot,\cdot,+,+,/,/\rangle$	1	$[[-10, 10], [0.1, 3]]$	4/1/3/0.0001	100/10/0.002
$x_0^2/2 + (x_1 + 1)^2/2$	$\langle\cdot,\cdot,+,+,/\rangle$	1, 2	$[[-20, -2], [2, 20]]$	3/1/2/0.1	150/5/0.005
		Program Functions			
$3x$ if $x > 0$, else x	$\langle\le,\le,\cdot,+,+,/\rangle$	1	$[-20, 20]$	2/1/1.5/0.1	100/5/0.005
x^2 if $x > 0$, else $-x$	$\langle\le,\le,-(\cdot),+,+,-,\cdot\rangle$	1	$[-20, 20]$	2/1/1.5/0.1	100/5/0.005
x if $x > 0$, else $\sin(x)$	$\langle\le,\le,+,+,\sin,\sin\rangle$	1	$[-20, 20]$	3/1/1.5/0.01	100/5/0.005
$\text{SORT}(x_0, x_1, x_2)$	$\langle\le,+,\text{MIN},\text{MAX},$ $\text{MAX}/,\cdot,-\rangle$	1, 2	$[-50, 50]^4$	3/1/4/0.01	100/5/0.004
$4\text{LFSR}(x_0, x_1, x_2, x_3)$	$\langle+,+,\text{XOR},\text{XOR}\rangle$	$\emptyset$	$\{0, 1\}^4$	2/1/1/0.1	100/5/0.005
$y_0(\vec{x}) = x_1$ if $x_0 < 2$, else $-x_1$ $y_1(\vec{x}) = x_0$ if $x_1 < 0$, else x_1^2	$\langle\le,\le,-(\cdot),\cdot\rangle$	1, 2	$[-5, 5]^2$	3/1/3/0.01	100/5/0.002
$g(x) = x^2$ if $x < 2$, else $x/2$ $y(x) = g^{\circ 4}(x)$	$\langle\le,\le,+,\cdot,\cdot,/,/\rangle$	1, 2	$[-8, 8]$	2/1/2/0.01	100/5/0.005
$g(x) = x + 2$ if $x < 2$, else $x - 1$ $y(x) = g^{\circ 2}(x)$	$\langle\le,\le,+,+,$ $+,-,-\rangle$	1, 2	$[-3, 6]$	2/1/1.5/0.01	100/5/0.005
		Pattern Recognition			
MNIST Binary	Φ_A	C	$[0, 1]^{784}$	2/1/10/0.01	150/ 10/0.05
MNIST Trinary	Φ_A	C	$[0, 1]^{784}$	2/1/10/0.01	150/ 10/0.05
ImageNet Binary*	Φ_A	C	$[0, 1]^{2048}$	4/1/10/10	150/10/0.0005
Backprop OccamNet*	Φ_B	C	$[0, 1]^{2048}$	4/1/10/NA	NA/NA/0.1
Finetune ResNet*	Φ_B	C	$[0, 1]^{3\times224\times224}$	4/1/10/NA	NA/NA/0.1

Table 4: Hyperparameters for Experiments Where $E = 1$

Target	Bases	Constants	Range	L	σ	R	λ	$w_\phi/w_\psi/w_\xi/w_\gamma$
		Analytic Functions						
$10.5x^3.1$	$\langle+,-,\cdot,/,\sin,$ $\cos,+c,\cdot c,(\cdot)^c\rangle$	$\emptyset$	$[0, 1]$	2	0.0005	200	10	0/0/0/0
$\cos(x)$	$\langle+,/,\sin\rangle$	$2, \pi$	$[-100, 100]$	3	0.01	400	50	0/0/0/0
e^x	$\langle+,\cdot c,(\cdot)^c\rangle$	10	$[0, 1]$	3	0.05	200	1	0.7/0.3/0.05/0.03
		Implicit Functions						
$x_0 x_1 = 1$	$\langle+,-,\cdot,/,\sin,\cos\rangle$	$\emptyset$	$[-1, 1]$	2	0.01	400	1	0.7/0.3/0.15/0.1
$x_0/x_1 = 1$	$\langle+,-,\cdot,/,\sin,\cos\rangle$	$\emptyset$	$[-1, 1]$	2	0.01	400	1	0.7/0.3/0.15/0.1
$x_0^2 + x_1^2 = 1$	$\langle+,-,\cdot,/,\sin,\cos\rangle$	$\emptyset$	$[-1, 1]$	2	0.01	200	10	0.7/0.3/0.15/0.1
$x_0/\cos(x_1) = 1$	$\langle+,-,\cdot,/,\sin,\cos\rangle$	$\emptyset$	$[-1, 1]$	2	0.01	200	10	0.7/0.3/0.15/0.1
$m_1 v_1 - m_2 v_2 = 0$	$\langle+,-,\cdot,/,\sin,\cos\rangle$	$\emptyset$	$[-10, 10]^3$	2	0.01	200	10	0.7/0.3/0.15/0.1

cannot be used as a one to one comparison with OccamNet. Hence, although we provide the raw data for XGB's performance, we do not analyze it further. We train all methods except "V100" on a single core of an Intel Xeon E5-2603 v4 @ 1.70GHz. For all methods, we use the basis set $\Phi = \langle+(\cdot,\cdot), -(\cdot,\cdot), \times(\cdot,\cdot), \div(\cdot,\cdot), \sin(\cdot), \cos(\cdot), \exp(\cdot), \log|\cdot|\rangle$.

For each dataset, we perform grid search to identify the best hyperparameters. The hyperparameters searched for the two OccamNet runs are shown in Table 6. The other hyperparameters not used in the grid search are set as follows: $T = 10$, $T_{\text{last}} = 10$, $w_\phi = w_\psi = w_\xi = w_\gamma = 0$, and the dataset batch size is the size of the training data. For OccamNet V100, we set R to be approximately as large as can fit on the V100 GPU, which varies between datasets. See Table 7 for the exact number of functions tested for each dataset for OccamNet V100. For XGBoost, we use exactly the same hyperparameter grid as used in Orzechowski et al. [53]. For Eplex, we use the same hyperparameter grid as used in Orzechowski et al. [53], with the exception that we use a depth of 4 to match that of OccamNet.

We select the best run from the grid search as follows. For each hyperparameter combination, we first identify the models with the lowest training MSE and the lowest validation MSE:

Table 5: Datasets Tested

#	Dataset	Size	# Features
1	1027_ESL	488	4
2	1028_SWD	1000	10
3	1029_LEV	1000	4
4	1030_ERA	1000	4
5	1089_USCrime	47	13
6	1096_FacultySalaries	50	4
7	192_vineyard	52	2
8	195_auto_price	159	15
9	207_autoPrice	159	15
10	210_cloud	108	5
11	228_elusage	55	2
12	229_pwLinear	200	10
13	230_machine_cpu	209	6
14	4544_GeographicalOriginalofMusic	1059	117
15	485_analcatdata_vehicle	48	4

Table 6: OccamNet Hyperparameters

Hyperparameter	OccamNet	OccamNet V100
α	$\{0.5, 1\}$	$\{0.5, 1\}$
σ	$\{0.5, 1\}$	$\{0.1, 0.5, 1\}$
E	$\{1, 5\}$	$\{0, 1, 5\}$
λ/R	$\{0.1, 0.5, 0.9\}$	$\{0.1, 0.5, 0.9\}$
R	$\{500, 1000, 2000\}$	max
N	$1000000/R$	1000

- For OccamNet, we examine the highest probability function after each epoch. From these functions, we select the function with the lowest testing MSE and the function with the lowest validation MSE.

- For Eplex, we examine the highest fitness individual from each generation. From these individuals, we select the individual with the lowest testing MSE and the individual with the lowest validation MSE.

- For XGBoost, we train the model until the validation loss has not decreased for 100 epochs. We then return this model as the model with the best training MSE and validation MSE.

Once we have the models with the lowest training and validation MSE for each hyperparameter combination, we identify the overall model with the lowest training MSE from the set of lowest

Table 7: Number of Functions Sampled

#	R
1	17123
2	8333
3	8333
4	8333
5	178571
6	166666
7	161290
8	52631
9	52631
10	78125
11	151515
12	41666
13	40000
14	7874
15	178571

Table 8: Raw data from the PMLB experiments. Hyperparameters and best fits are in the following path in our code (see Section M): `pmbl-experiments/pmlb-results`.

Training Loss (MSE)						Validation Loss (MSE)				
#	OccamNet	V100	Eplex	AIF	XGB	OccamNet	V100	Eplex	AIF	XGB
1	0.177	**0.139**	0.153	0.465	0.056	0.141	0.137	**0.128**	0.449	0.133
2	0.607	**0.605**	0.643		0.443	0.647	**0.640**	0.702		0.567
3	0.486	**0.432**	0.443		0.326	0.634	0.597	**0.581**		0.556
4	0.639	0.616	**0.616**	0.886	0.547	0.662	0.641	**0.641**	1.040	0.649
5	0.107	**0.054**	0.105		0.000	0.145	**0.108**	0.182		0.134
6	0.070	**0.035**	0.067	0.162	0.000	0.037	**0.017**	0.036	0.066	0.114
7	0.228	**0.161**	0.230	0.713	0.039	0.047	**0.099**	0.122	0.802	0.175
8	0.155	**0.145**	0.152		0.000	0.095	0.115	**0.097**		0.105
9	0.168	**0.141**	0.152		0.000	0.114	**0.097**	0.129		0.105
10	0.154	**0.101**	0.130	0.171	0.000	0.027	**0.021**	0.036	0.044	0.162
11	0.136	**0.129**	0.141	0.177	0.029	0.119	0.162	**0.161**	0.178	0.106
12	0.255	**0.167**	0.324		0.000	0.193	**0.177**	0.310		0.083
13	0.062	**0.042**	0.082	0.103	0.004	0.074	**0.076**	0.198	0.289	0.163
14	0.573	0.438	**0.414**		0.000	0.470	**0.312**	0.320		0.196
15	0.208	**0.183**	0.216	0.456	0.000	0.174	**0.411**	0.567	0.524	0.175

Testing Loss (MSE)						Average Run Time (s)				
#	OccamNet	V100	Eplex	AIF	XGB	OccamNet	V100	Eplex	AIF	XGB
1	0.251	**0.141**	0.223	0.741	0.147	2246	319	4574	4498	5
2	0.704	**0.706**	0.742		0.648	4789	302	4651		7
3	0.542	0.491	**0.474**		0.464	4767	300	4696		6
4	0.713	0.679	**0.679**	0.895	0.659	4511	308	4729	4222	5
5	0.589	0.209	**0.116**		0.113	239	504	4684		3
6	0.151	**0.082**	0.091	0.088	0.234	244	479	4755	5076	3
7	0.335	0.761	0.829	**0.698**	0.316	249	468	4583	1222	1
8	0.137	0.132	**0.119**		0.094	751	359	4516		5
9	0.135	0.194	**0.150**		0.094	764	354	4517		5
10	0.084	**0.086**	0.097	0.109	0.080	483	373	4605	6196	2
11	0.180	**0.177**	0.275	0.274	0.150	259	455	4639	1223	2
12	0.223	**0.214**	0.426		0.165	944	335	4653		3
13	1.088	**0.157**	0.487	0.864	0.622	956	364	4561	6899	5
14	0.570	**0.440**	0.442		0.228	6562	354	4785		139
15	1.664	**0.334**	0.441	0.324	0.188	264	487	4533	586	29

training MSE models and we identify the overall model with the lowest validation MSE from the set of lowest validation MSE models. We then record these models' training MSE and validation MSE as the best training MSE and validation MSE, respectively. Finally, we test the model with the overall lowest validation MSE on the testing dataset, and record the result as the grid search testing MSE.

The raw data from this experiment is shown in Table 8. To improve readability, we use red highlighting and bold text to illustrate the best performing model for each dataset and metric. We compare OccamNet, Eplex, and AIF, marking the method with the lowest MSE or training time in red. We also compare V100, Eplex, and AIF, marking the method with the lowest MSE or training time in bold.

As discussed in the main text, OccamNet is considerably faster than Eplex, often running faster by more than an order of magnitude. This may be in part because we train Eplex with the DEAP evolutionary computation framework [51], which is implemented in Python and utilizes NumPy arrays for computation. Thus, our implementation of Eplex may be somewhat slower than an implementation written in C. However, because of its selection based on many fitness cases, Eplex is also by nature considerably slower than many other genetic algorithms, running in $O(TN^2)$, where T is the number of fitness cases and N is the population size [58]. This suggests that even a C implementation of Eplex may not be as fast as OccamNet. More recent selection algorithms perform comparably to Eplex but run significantly faster, for example Batch Tournament Selection [58]. However, because these methods did not exist at the time of Orzechowski et al. [53], they have not been compared to other methods on the PMLB datasets. Thus, we have not tested these methods here. On the other hand, our current implementation of sampling and the forward pass work with

DAGs in which an edge leads to each argument node, regardless of whether the argument node is connected to the outputs. The result is that our implementation of OccamNet evaluates more than $|\Phi|$ times more basis functions than is necessary, where $|\Phi|$ is the number of basis functions. In the case of these experiments, this amounts to more than 8 times the number of calculations necessary. In the future, we plan to implement OccamNet by constructing DAGs starting from the output nodes, thereby greatly increasing speed.

I Analysis of Fits to PMLB Datasets

In this section, we analyze the fits that the methods discussed in Section H provide for the PMLB dataset.

OccamNet's solutions are all short, easy to comprehend fits to the data. We find that OccamNet uses addition, subtraction, multiplication, and division most extensively, exploiting $\sin(\cdot)$ and $\cos(\cdot)$ for more nonlinearity. Interestingly, OccamNet uses $\exp(\cdot)$ and $\log|\cdot|$ less frequently, perhaps because both functions can vary widely with small changes in input, making functions with these bases more likely to represent poor fits.

OccamNet's solutions demonstrate its ability to exploit modularity and reuse components. These solutions often have repeated components, for example in dataset #1, 1027_ESL, where the best fit to the training data is

$$y_0 = \frac{(\sin(x_2) + x_3 + x_1) \cdot (\sin(x_2) + x_3 + x_1)}{(\sin(x_2) + x_3 + x_1) + (x_3 + x_1) + (x_1 + x_3)}.$$

In this fit, OccamNet builds $\sin(x_2) + x_3 + x_1$ in the first two layers of the network and then reuses it three times. Solutions like the above demonstrate OccamNet's ability to identify successful subcomponents of a solution and then to rearrange the subcomponents into a more useful form. Examples like the above, however, also demonstrate that OccamNet often overuses modularity, potentially restricting the domain of functions it can search. We suspect that the main reason that OccamNet may rely too heavily on modularity in some fits is that OccamNet uses an extremely high learning rate of 1 for its training. We used such a large learning rate to allow OccamNet to converge even when faced with 10^{30} or more functions. However, we suspect that this may also cause OccamNet to converge to certain paths before exploring sufficiently. For example, with the function above, OccamNet may have identified that $\sin(x_2) + x_3 + x_1$ is a useful component and, because of its high learning rate, used this pattern in several times instead of the one time needed. This hypothesis is supported by the fact that OccamNet V100, which samples many more functions before taking a training step, repeats patterns less frequently than OccamNet. For example OccamNet V100's best fit solution for the training dataset of dataset #1 is

$$y_0 = \cos(x_1/x_1) \cdot \cos(x_1/x_1) \cdot (x_2 + x_3 + \sin(x_0) + x_3 + x_1 - \sin(x_3)),$$

which contains almost no repetition.

Remarkably, for dataset #4, 1030_ERA, both Eplex and OccamNet V100 discover equivalent functions for both training and validation: OccamNet V100 discovers

$$y_0 = \cos(\sin(x_1 - x_2)) \cdot (\sin(x_2) + x_0 + x_1) \cdot \cos(x_2/x_2),$$

and Eplex discovers

$$y_0 = \cos(x_1/x_1) \cdot (\sin(x_2) + x_0 + x_1) \cdot \cos(\sin(x_2 - x_1)).$$

As a result, the two methods' losses are identical up to 7 decimal places. Still, we mark Eplex as performing better on this dataset because after the 7th decimal place it has a slightly lower loss, likely due to differences in rounding or precision between the two approaches. Two different methods identifying the same function is extremely unlikely; OccamNet's search space includes $2 \cdot 10^{30}$ paths for this dataset, meaning that the probability of both methods identifying this function purely by chance is miniscule. This, in combination with the fact that this function was the best fit to both the training and validation datasets for both methods, suggests that the identified function is a nearly optimal fit to the data for the given search space. Given the size of the search space, this result thus provides further evidence that OccamNet and Eplex perform far better than brute force search. Interestingly, although OccamNet did not discover this function, it's best fit for the validation,

$$y_0 = \sin(x_2/x_2) \cdot (\sin(x_2) + x_0 + x_1) \cdot \cos(\cos(x_3)) \cdot \cos(\sin(x_3)),$$

does include several features present in the fits found by V100 and Eplex, such as the the $\sin(x_2) + x_0 + x_1$ term, the $\cos(\sin(\cdot))$ term, and the x_2/x_2 inside of the trigonometric function. This suggests that OccamNet may also have been close to converging to the function discovered by Eplex and OccamNet V100. OccamNet's loss was also always within 5% of Eplex's loss on this dataset, again suggesting that OccamNet had identified a function close to that of Eplex and V100.

Interestingly, AI Feynman 2.0's fits generally tend to be very simple compared to those of OccamNet. For example, AIF's fit for the training dataset #11 is

$$y_0 = -0.050638447726 + \log(x_0/\sin(x_0)) - x_0,$$

whereas OccamNet's fit is

$$y_0 = \sin(x_0) \cdot x_1 \cdot x_0 \cdot \sin(x_0) \cdot \log|x_0| - \cos(x_1 \cdot x_0 - x_1).$$

AI Feynman's fit is slightly simpler and easier to interpret, but it comes at the cost of having a 35% higher loss. We suspect that because the PMLB datasets likely do not have modular representations, AI Feynman must rely mainly on its brute force search, which ultimately produces shorter expressions. AI Feynman can also produce constants because of its polynomial fits, and it uses constants in nearly every solution it proposes. We did not allow the other symbolic methods to fit constants, but they still consistently performed better than AI Feynman, suggesting that fitting constants may not be essential to accurately modeling the PMLB datasets.

J   Ablation Studies

We test the performance of each hyperparameter in a collection of ablation studies, as shown in Table 9. Here, we focus on what our experiments demonstrate to be the most critical parameters to be tuned: the collection of bases and constants, the network depth, the variance of our interpolating function, the overall network temperature (as well as the last layer temperature), and, finally, the learning rate of our optimizer. As before, we set the stop criterion and terminate learning when the top-λ sampled functions all return the same fitness $K(\cdot, f)$ for 30 consecutive epochs. If this does not occur in a predefined, fixed number of iterations, or if the network training terminates and the final expression does not match the correct function we aim to fit, we say that the network has not converged. All hyperparameters for baselines are specified in Section G, except for the sampling size, which is set to $R = 100$.

Our benchmarks use a sampling size large enough for convergence in most experiments. It is worth noting, however, that deeper networks failed to always converge (with convergence fraction of $\eta = 8/10$) for the analytic function we tested. Deeper networks allow for more function composition and let approximations emerge as local minima: in practice, we find that increasing the last layer temperature or reducing the variance is often needed for allowing for a larger depth L. For pattern recognition, we found that *MNIST Binary* and *Trinary* require depth 2 for successful convergence, while the rest of the experiments require depth 4. Shallower or deeper networks either yield subpar accuracy or fail to converge. We also find that for OccamNet without skip connections, larger learning rates usually work best, i.e. 0.05 works best, while OccamNet with skip connections requires a smaller learning rate, usually around 0.0005. We also tested different temperature and variance schedulers, in the spirit of simulated annealing. In particular, we tested increasing or decreasing these parameters over training epochs, as well as sinusoidally varying them with different frequencies. Despite the increased convergence time, however, we did not find any additional benefits of using schedulers. As we test OccamNet in larger problems spaces, we will revisit these early scheduling studies and investigate their effects in those domains.

K   Neural Approaches to Benchmarks

Since our OccamNet is a neural model that is constructed on top of a fully connected neural architecture, below we first consider a limitation of the standard fully connected architectures for sorting, and then a simple application of our temperature-controlled connectivity.

A   Exploring the limits of fully connected neural architectures for sorting

We made a fully connected neural network with residual connections. We used the mean squared error (MSE) as the loss function. The output size was equal to the input size and represented the

Table 9: Ablation studies on representative experiments

Modification	Convergence fraction η	Convergence epochs T_c
Experiment $\sin(3x + 2)$		
baseline	10/10	390
added constants (2) and bases $(\cdot, (\cdot)^2, -(\cdot))$	10/10	710
lower last layer temperature (0.5)	10/10	300
higher last layer temperature (3)	10/10	450
lower learning rate (0.001)	10/10	2500
higher learning rate (0.01)	10/10	170
deeper network (6)	8/10	3100
lower variance (0.0001)	10/10	390
higher variance (0.1)	10/10	450
lower sampling (50)	10/10	680
higher sampling (250)	10/10	200
Experiment x^2 if $x > 0$, else $-x$		
baseline	10/10	100
added constants (1, 2) and bases $(-, -(\cdot))$	10/10	290
lower last layer temperature (0.5)	10/10	160
higher last layer temperature (3)	10/10	150
lower learning rate (0.001)	10/10	780
higher learning rate (0.01)	10/10	90
deeper network (6)	10/10	180
shallower network (2)	10/10	160
lower variance (0.001)	10/10	160
higher variance (1)	10/10	180
lower sampling (50)	10/10	290
higher sampling (250)	10/10	140

original numbers in a sorted order. We used L_2 regularization along with Adam optimization. We tested weight decay ranging from 1e-2 to 1e-6 in which 1e-5 provided the best training and testing accuracy. Finally learning rate for the optimiser was found to be optimum around 1e-3. We used $30,000$ data points to train the model with batch size of 200. Each of the data point was a list of numbers between 0 and 100. For a particular value of input size x (representing number of points to be sorted), the number of hidden units was varied from 2 to 20 and the number of hidden layers was varied from 2 (just a input and a output layer) to $x! + 2$. Then, the test loss was calculated on 20,000 points, chosen from same distribution. Finally, the combination (hidden_layer, hidden_unit) for which the loss was less than 5 and (hidden_ layer * hidden_units) was min was noted in Table 10. As seen from the table, the system failed to find any optimal combination for any input size greater than or equal to 5. For example, for input size 5, the hidden units were upper capped at 20 and hidden layers at 120 and thus 2400 parameters were insufficient to sort 5 numbers.

Table 10: Minimal configurations to sort list of length "input size."

Input Size	Hidden units	Hidden Layers	Parameters
2	6	2	12
3	8	4	32
4	18	4	72
5	-	-	-

Generalization The model developed above generalizes poorly on data outside the training domain. For example, consider the model with 18 hidden units and 4 hidden layers, which is successfully trained to sort 4 numbers chosen from range 0 to 100. It was first tested on numbers from 0 to 100 and then on 100 to 200. The error in the first case was around 2 while the average error in the second case was between 6 and 8 (which is $(200/100)^2 = 4$ times the former loss). Finally when tested on larger ranges such as $(9900, 10000)$, the error exploded to around 0.1 million (which is an order

greater than $(10000/100)^2 = 10000$ times the original loss). This gives a hint that the error might be scaling proportionally to the square of test domain with respect to the train domain. A possible explanation of this comes from the use of MSE as loss function. Scaling test data by ρ scales the absolute error by approximately the same factor and then taking a square of the error to calculate the MSE scales the total loss by square of that factor, i.e., ρ^2.

B Applying temperature-controlled connectivity to standard neural networks for MNIST classification

We would like to demonstrate the promise of the temperature-controlled connectivity as a regularization method for the classification heads of models with a very simple experiment. We used the ResNet50 model to train on the standard MNIST image classification benchmark. We studied two variants of the model: one is the standard ResNet model and the other is the same model, but augmented with our temperature-controlled connectivity (with $T = 1$) between the flattened layer and the last fully connected layer (on the lines discussed in the main paper). Then we trained both models with a learning rate fixed at 0.05 and a batch size of 64 and ran it for 10 epochs. The model with regularization performed slightly better than one without it. The regularized model achieved the maximum accuracy of 99.18% while the same figure for the standard one was 98.43%. Besides, another interesting observation that we made was about the stability of the results. The regularized model produced much more stable and consistent results across iterations as compared to one without it. These results encourage us to study the above regularization method in larger experiments.

L Symbolic Regression Benchmarks

A Eureqa

Eureqa is a software package for symbolic regression where one can specify different target expressions, building block functions (analogous to the bases in OccamNet), and loss functions [28]. For most functions, we use the absolute error as the optimization metric. We choose formula building blocks in Eureqa to match the bases functions used in OccamNet.

For implicit functions, we use the implicit derivative error. We also order the data to improve the performance. For the implicit functions in lines 1, 3, and 4 in Table 2 of the main text, the data is ordered by x_0. For the equation $x_0^2 + x_1^2 = 1$, the data is generated by sampling $\theta \in [0, 2\pi)$ and calculating $x_0 = \sin(x)$ and $x_1 = \cos(x)$, and is ordered by θ. When the data is not ordered the value of the implicit derivative error is much higher, resulting in the algorithm favoring incorrect equations. For equation $m_1 v_1 - m_2 v_2$, the ordering is more ambiguous because of the higher dimensionality. We tried ordering by both m_1 and the product $m_1 v_1$ without success.

B HeuristicLab

Due to limits on the number of data points and feature columns in Eureqa, we instead use HeuristicLab for the image recognition tasks described in Section 5.4 of the main text. HeuristicLab is a software package for optimization and evolutionary algorithms, and includes symbolic regression and symbolic classification. We use the Island Genetic Algorithm with default settings.

Similar to the building block functions in Eureqa, HeuristicLab can specify the basis symbols for each task. However, HeuristicLab does not have the bases MAX, SIGMOID, or tanh. Instead we use the symbols IfThenElse, GreaterThan, LessThan, And, Or, and Not.

C Eplex

As discussed in Section H, Eplex [52], short for Epsilon-Lexicase selection, is a genetic programming population selection technique which we use as a symbolic regression benchmark in our experiments with PMLB datasets. We implement a genetic algorithm using Eplex with the DEAP [51] evolutionary framework, using Numpy arrays [59] for computation to increase speed.

Eplex selects individuals from a population by evaluating the individuals on subsets, or fitness cases, of the full data. For a given fitness case, Eplex selects the top performing individuals, and then

proceeds to the next fitness case. This process is repeated until only one individual remains. This individual is then used as the parent for the next generation.

D AI Feynman 2.0

We also benchmark OccamNet against AI Feynman 2.0 [54]. AI Feynman 2.0 is a mixed approach that combines brute force symbolic regression, polynomial fits, and identification for modularity in the data using neural networks. To identify modularities in the data, AI Feynman first trains a neural network on it. This serves as an interpolating function for the true data, and allows the network to search for symmetries and other forms of modularities.
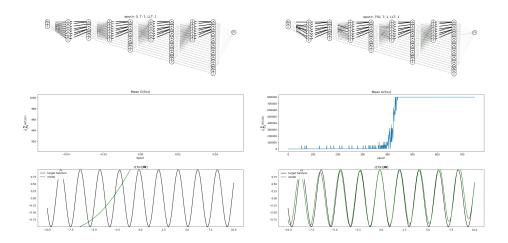

Figure 5: In this figure we present two video frames for the target $\sin(3x + 2)$, which could be accessed via `videos/sin(3x + 2).mp4` in our code files. We show the beginning of the fitting (left) and the end, where OccamNet has almost converged (right).
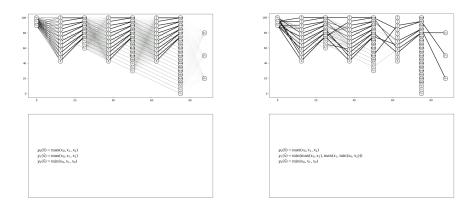

Figure 6: In this figure we present two video frames for the target $\text{SORT}(x_0, x_1, x_2)$, which could be accessed via `videos/sorting.mp4` in our code files. We show the beginning of the fitting (left) and the end, where OccamNet has almost converged (right).

M Code

We will make our code public.   We have grouped our code into five main folders. `analytic-and-programs` stores our network and experiments for fitting analytic functions and

programs. `implicit` stores our network and experiments for implicit functions, although it also includes the three analytic functions listed in Table 4. `constant-fitting` stores code very similar to `implicit` but which is optimized for constant fitting. `image-recognition` stores our network and experiments for image classification. `pmlb-experiment` stores our code for our benchmarking against the PMLB regression datasets. Finally, `videos` stores several videos of our model converging to various functions. In Figures 5 and 6 we present snapshots of the videos.

References

[1] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.

[2] Anna Choromanska, Mikael Henaff, Michael Mathieu, Gérard Ben Arous, and Yann LeCun. The loss surfaces of multilayer networks. In *AISTATS*, 2015.

[3] Guillaume Lample and François Charton. Deep learning for symbolic mathematics. In *ICLR*, 2020.

[4] Riccardo Poli, William B Langdon, Nicholas F McPhee, and John R Koza. *A field guide to genetic programming*. lulu.com, 2008.

[5] P. J. Angeline, G. M. Saunders, and J. B. Pollack. An evolutionary algorithm that constructs recurrent neural networks. *IEEE Transactions on Neural Networks*, 5(1):54–65, 1994.

[6] Dirk V. Arnold and Nikolaus Hansen. A (1+1)-CMA-ES for constrained optimisation. In *GECCO*, 2012.

[7] Felipe Petroski Such, Vashisht Madhavan, Edoardo Conti, Joel Lehman, Kenneth O. Stanley, and Jeff Clune. Deep neuroevolution: Genetic algorithms are a competitive alternative for training deep neural networks for reinforcement learning. *arXiv preprint arXiv:1712.06567*, 2017.

[8] Nikolaus Hansen. The CMA evolution strategy: A tutorial. *arXiv preprint arXiv:1604.00772*, 2016.

[9] Ilya Loshchilov and Frank Hutter. CMA-ES for hyperparameter optimization of deep neural networks. *arXiv preprint arXiv:1604.07269*, 2016.

[10] Martin Fowler. *Domain Specific Languages*. Addison-Wesley Professional, 1st edition, 2010.

[11] Georg Martius and Christoph Lampert. Extrapolation and learning equations. *arXiv preprint arXiv:1610.02995*, 2016.

[12] Subham Sahoo, Christoph Lampert, and Georg Martius. Learning equations for extrapolation and control. In *ICML*, 2018.

[13] Samuel Kim, Peter Y. Lu, Srijon Mukherjee, Michael Gilbert, Li Jing, Vladimir Čeperić, and Marin Soljačić. Integration of neural network-based symbolic regression in deep learning for scientific discovery. *IEEE Transactions on Neural Networks and Learning Systems*, pages 1–12, 2020.

[14] Christos Louizos, Max Welling, and Diederik P. Kingma. Learning sparse neural networks through L_0 regularization. In *ICLR*, 2018.

[15] Zongben Xu, Hai Hong Zhang, Yao Wang, Xiangyu Chang, and Yong Liang. L1/2 regularization. *Science China Information Sciences*, 53:1159–1169, 2010.

[16] Trevor Gale, Erich Elsen, and Sara Hooker. The state of sparsity in deep neural networks. *arXiv preprint arXiv:1902.09574*, 2019.

[17] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

[18] Dimitris Bertsimas, John Tsitsiklis, et al. Simulated annealing. *Statistical science*, 8(1):10–15, 1993.

[19] Chris J. Maddison, Andriy Mnih, and Yee Whye Teh. The concrete distribution: A continuous relaxation of discrete random variables. In *ICLR*, 2017.

[20] Eric Jang, Shixiang Gu, and Ben Poole. Categorical reparameterization with gumbel-softmax. In *ICLR*, 2017.

[21] George Tucker, Andriy Mnih, Chris J. Maddison, and Jascha Sohl-Dickstein. REBAR: low-variance, unbiased gradient estimates for discrete latent variable models. In *NIPS*, 2017.

[22] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. Densely connected convolutional networks. In *CVPR*, 2017.

[23] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *CVPR*, pages 770–778, 2016.

[24] Ronald J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Mach. Learn.*, 8(3–4):229–256, May 1992.

[25] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *ICLR*, 2015.

[26] Andrew Trask, Felix Hill, Scott E Reed, Jack Rae, Chris Dyer, and Phil Blunsom. Neural arithmetic logic units. In *NIPS*. 2018.

[27] Andreas Madsen and Alexander Rosenberg Johansen. Neural arithmetic units. In *ICLR*, 2020.

[28] Michael Schmidt and Hod Lipson. Distilling free-form natural laws from experimental data. *Science*, 324(5923):81–85, 2009.

[29] Silviu-Marian Udrescu and Max Tegmark. AI Feynman: A physics-inspired method for symbolic regression. *Science Advances*, 6(16), 2020.

[30] Robert I Mckay, Nguyen Xuan Hoai, Peter Alexander Whigham, Yin Shan, and Michael O'neill. Grammar-based genetic programming: a survey. *Genetic Programming and Evolvable Machines*, 11(3-4):365–396, 2010.

[31] Brenden K Petersen, Mikel Landajuela Larma, Terrell N. Mundhenk, Claudio Prata Santiago, Soo Kyung Kim, and Joanne Taery Kim. Deep symbolic regression: Recovering mathematical expressions from data via risk-seeking policy gradients. In *ICLR*, 2021.

[32] Kevin Ellis, Daniel Ritchie, Armando Solar-Lezama, and Josh Tenenbaum. Learning to infer graphics programs from hand-drawn images. In *NIPS*. 2018.

[33] Kevin Ellis, Lucas Morales, Mathias Sablé-Meyer, Armando Solar-Lezama, and Josh Tenenbaum. Learning libraries of subroutines for neurally–guided bayesian program induction. In *NIPS*. 2018.

[34] Kevin Ellis, Maxwell Nye, Yewen Pu, Felix Sosa, Josh Tenenbaum, and Armando Solar-Lezama. Write, execute, assess: Program synthesis with a REPL. In *NeurIPS*. 2019.

[35] Alex Graves, Greg Wayne, and Ivo Danihelka. Neural turing machines. *arXiv preprint arXiv:1410.5401*, 2014.

[36] Alex Graves, Greg Wayne, Malcolm Reynolds, Tim Harley, Ivo Danihelka, Agnieszka Grabska-Barwinska, Sergio Gómez Colmenarejo, Edward Grefenstette, Tiago Ramalho, John Agapiou, Adrià Puigdomènech Badia, Karl Moritz Hermann, Yori Zwols, Georg Ostrovski, Adam Cain, Helen. King, C. Summerfield, Phil Blunsom, Koray Kavukcuoglu, and Demis Hassabis. Hybrid computing using a neural network with dynamic external memory. *Nature*, 538:471–476, 2016.

[37] Mark Collier and Joeran Beel. Implementing neural turing machines. In *ICANN*, page 94–104, 2018.

[38] Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. Deepcoder: Learning to write programs. In *ICLR*, 2016.

[39] Armando Solar Lezama. *Program Synthesis By Sketching*. PhD thesis, EECS Department, University of California, Berkeley, 2008.

[40] Karol Kurach, Marcin Andrychowicz, and Ilya Sutskever. Neural random-access machines. In *ICLR*, 2016.

[41] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. In *IEEE*, 1998.

[42] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR*, 2009.

[43] David J. Montana and Lawrence Davis. Training feedforward neural networks using genetic algorithms. In *IJCAI*. Morgan Kaufmann Publishers Inc., 1989.

[44] Tim Salimans, Jonathan Ho, Xi Chen, Szymon Sidor, and Ilya Sutskever. Evolution strategies as a scalable alternative to reinforcement learning. *arXiv preprint arXiv:1703.03864*, 2017.

[45] Edouard Grave, Armand Joulin, Moustapha Cissé, David Grangier, and Hervé Jégou. Efficient softmax approximation for GPUs. In *ICML*, 2017.

[46] Song Han, Jeff Pool, John Tran, and William J. Dally. Learning both weights and connections for efficient neural networks. 2015.

[47] Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. Pruning filters for efficient convnets. In *ICLR*, 2017.

[48] Dmitry Molchanov, Arsenii Ashukha, and Dmitry Vetrov. Variational dropout sparsifies deep neural networks. In *ICML*, 2017.

[49] Stefan Wagner, Gabriel Kronberger, Andreas Beham, Michael Kommenda, Andreas Scheibenpflug, Erik Pitzer, Stefan Vonolfen, Monika Kofler, Stephan Winkler, Viktoria Dorfer, and Michael Affenzeller. *Advanced Methods and Applications in Computational Intelligence*, volume 6, chapter Architecture and Design of the HeuristicLab Optimization Environment, pages 197–261. Springer, 2014.

[50] Randal S. Olson, William La Cava, Patryk Orzechowski, Ryan J. Urbanowicz, and Jason H. Moore. PMLB: a large benchmark suite for machine learning evaluation and comparison. *BioData Mining*, 10(1):36, 2017.

[51] Félix-Antoine Fortin, François-Michel De Rainville, Marc-André Gardner, Marc Parizeau, and Christian Gagné. DEAP: Evolutionary algorithms made easy. *Journal of Machine Learning Research*, 13:2171–2175, 2012.

[52] William La Cava, Lee Spector, and Kourosh Danai. Epsilon-Lexicase Selection for Regression. In *GECCO*, 2016.

[53] Patryk Orzechowski, William La Cava, and Jason H. Moore. Where are we now? A large benchmark study of recent symbolic regression methods. In *GECCO*, 2018.

[54] Silviu-Marian Udrescu, Andrew Tan, Jiahai Feng, Orisvaldo Neto, Tailin Wu, and Max Tegmark. AI Feynman 2.0: Pareto-optimal symbolic regression exploiting graph modularity. 2020.

[55] Michael Schmidt and Hod Lipson. *Symbolic Regression of Implicit Equations*, pages 73–85. Springer US, 2010.

[56] Aaron Meurer, Christopher P. Smith, Mateusz Paprocki, Ondřej Čertík, Sergey B. Kirpichev, Matthew Rocklin, AMiT Kumar, Sergiu Ivanov, Jason K. Moore, Sartaj Singh, Thilina Rathnayake, Sean Vig, Brian E. Granger, Richard P. Muller, Francesco Bonazzi, Harsh Gupta, Shivam Vats, Fredrik Johansson, Fabian Pedregosa, Matthew J. Curry, Andy R. Terrel, Štěpán Roučka, Ashutosh Saboo, Isuru Fernando, Sumith Kulal, Robert Cimrman, and Anthony Scopatz. SymPy: symbolic computing in python. *Peer J Computer Science*, 3:103, 2017.

[57] Tianqi Chen and Carlos Guestrin. XGBoost: A Scalable Tree Boosting System. *arXiv e-prints*, art. arXiv:1603.02754, March 2016.

[58] Vinicius V. Melo, Danilo Vasconcellos Vargas, and Wolfgang Banzhaf. Batch Tournament Selection for Genetic Programming. In *GECCO*, 2019.

[59] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, 2020.