Stochastic optimization approaches to synthesis of intepretable representations

Anonymous Author(s)
Affiliation
Address
email

Abstract

We propose a method for learning interpretable features using stochastic optimization approaches to optimize feature structures. Features are represented as multi-type expression trees using a set of activation functions common in neural networks in addition to other elementary functions. Continuous features are trained via backpropagation, and the performance of features in ML models is used to weight the rate of change among representations. We compare several stochastic optimization approaches to searching the architectures of these features. The search process maintains an archive of representations with accuracy-complexity trade-offs to assist in generalization and interpretation. We benchmark the results against other machine learning approaches suggest this is a worthwhile approach for finding simpler models of various processes.

1 Introduction

The performance of a machine learning (ML) model depends primarily on the data representation used in training [2], and for this reason the representational capacity of neural networks (NN) is considered a central factor in their success in many applications [4]. To date, there does not seem to be a consensus on how ϕ should be designed. As problems grow in complexity, the networks proposed to solve these problems grow as well, leading to an intractable hypothesis space for ϕ. One approach is to tune ϕ through network hyperparameters using grid search or randomized search with cross validation. Another is to use population-based search with stochastic optimization (SO), which is the focus of this paper. In SO, several candidate architectures are evaluated and varied over several iterations, and a heuristic is used to probabilistically select and update them until the population produces an adequate architecture.

In practice, the adequacy of the architecture, i.e. model *form*, is often dependent on conflicting objectives. Interpretability is a central focus in machine learning (ML) research, because many researchers in the scientific community rely on ML models not only to provide predictions that match data from various processes, but to provide insight into the nature of the processes themselves. Approaches to interpretability can be roughly grouped into semantic and syntactic approaches. Semantic approaches encompass methods that attempt to illucidate the behavior of a model under various input conditions as a way of explanation. Syntactic methods instead focus on the development of concise models that, by virtue of their simplicity, can be interpreted in a similar vein to first-principles models. The method proposed here belongs to the latter group: our goal is to discover the simplest descriptions of a process whose predictions generalize as well as possible.

Building blocks of good solutions are propagated by evolutionary computation via mutation and recombination of sub-functions. This promotes functional modularity, which is a key to evolvability and also key to distributed representations, in which we desired networks to solve modular tasks. Modularity is important for interpretability as well.

Submitted to 32nd Conference on Neural Information Processing Systems (NIPS 2018). Do not distribute.

Disentangling factors of variation [2] could certainly assist in interpretation.

In our case we are interested in discovering the simplest set of constructed features that generalize well. Since the ideal trade-off between generalization and interpretability is not known *a priori*, it is useful to characterize the Pareto front, i.e. the set of models for which any improvement in interpretability or accuracy worsens the other objective. Stochastic methods are useful in this regard.

2 Methods

We are interested in the tasks of regression and classification, for which the goal is to build a predictive model $\hat{y}(\mathbf{x})$ using N paired examples $\mathcal{T} = \{(\mathbf{x}_i, y_i)\}_{i=1}^N$. For regression, $\hat{y}(\mathbf{x})$ associates the inputs $\mathbf{x} \in \mathbb{R}^d$ with a real-valued output $y \in \mathbb{R}$. For classification, $\hat{y}(\mathbf{x})$ instead maps $\mathbf{x}$ to one of k class labels from the set $\mathcal{Y} = \{1 \ \ldots \ k\}$, with labels $y \in \mathcal{Y}$.

The goal of feature engineering / representation learning is to find a new representation of $\mathbf{x}$ via a m-dimensional feature mapping $\mathbf{\Phi}(\mathbf{x}) : \mathbb{R}^d \rightarrow \mathbb{R}^m$, such that a model $\hat{y}(\mathbf{\Phi}(\mathbf{x}))$ outperforms the model $\hat{y}(\mathbf{x})$.

When applying a NN to a traditional ML task like regression or classification, a fixed NN architecture $\phi(\mathbf{x}, \theta)$, parameterized by θ, is designed by expertise or manual tuning, and used to fit a model

$$\hat{y} = \phi(\mathbf{x}, \theta)^T \beta \tag{1}$$

In this case $\phi = [\phi_1 \ \ldots \ \phi_m]^T$ is a NN with m nodes in the hidden layer and a linear output layer with coefficients $\beta = [\beta_1 \ \ldots \ \beta_m]^T$ produces the model output. The problem is then cast as a (non-convex) parameter optimization problem of the form

$$\theta^* = \arg\min_{\theta} \sum_{i}^{N} L(y_i, \hat{y}_i, \theta, \beta) \tag{2}$$

where $\hat{\theta}$ is chosen to minimize a cost function L, with global optimum θ^*. (Note L may also depend on θ and β in the case of regularization.)

In the case of SO, the optimization problem is made more general to include the form of ϕ in the minimization fo L, leading to the formulation

$$\phi^*(\mathbf{x}, \theta^*) = \arg\min_{\phi \in \mathbb{S}, \theta} \sum_{i}^{N} L(y_i, \hat{y}_i, \phi, \theta, \beta) \tag{3}$$

where $\mathbb{S}$ is the space of possible functions defined by the procedure, and ϕ^* is the true structure of the process underlying the data. The assumption of stochastic optimization approaches such as evolutionary computation (EC) and simulated annealing (SA) is that candidate solutions in $\mathbb{S}$ that are similar to each other, i.e. reachable in few mutations, are more likely to have similar costs than candidate solutions that are far apart (an assumption known as locality). In these cases, $\mathbb{S}$ can be effectively searched by maintaining and updating a population of candidate representations that perform well.

The method described in this paper is called the feature engineering archiving tool (Feat). It is a wrapper-based feature synthesis technique that optimizes features in the loop with a user-given ML method. FEAT contributes a few non-standard SO techniques designed to improve the generalizability and legibility of the method. First, it incorporates the initial ML model into the population during training. Second, it fits an ML model to each candidate representation to a) assess its fitness and b) provide feedback to the variation process at a more granular level. Third, it maintains an archive of Pareto optimal trade-offs between complexity and accuracy. This archive is validated on a hold-out test set at the end of training to conduct model/representation selection.

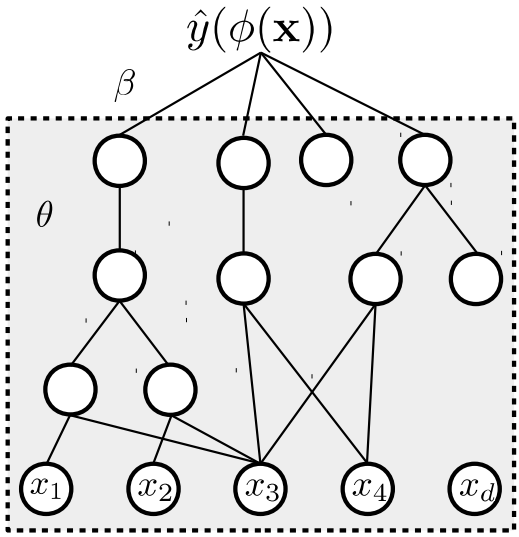

Figure 1: Example model representation in FEAT.

Table 1: Functions and terminals used to develop representations.

Continuous functions	{+, -, *, /, ^{2}, ^{3}, $\sqrt{}$, sin, cos, exp, log, exponent, logit, tanh, gauss, relu}
Boolean functions	and, or, not, xor, =, <, <=, >, >=
Terminals	{$\mathbf{x}$}

2.1 Feat

Feat uses a typical evolutionary updating scheme to optimize representations. A population of potential representations, $\mathcal{N} = \{n_1 \ \ldots \ n_P\}$, is constructed uniform randomly from a set of continuous and boolean functions and terminals. Each representation is used to fit an ML model. For the experiments in this paper, we use linear ridge regression for regression and logistic regression with regularization for classification. Following this initial forward pass, the weights of the differentiable programs are updated using stochastic gradient descent with backpropagation. The ML model's weights are then used to bias the variation of each individual.

2.2 Representation

Feat represents features by constructing syntax trees from elementary boolean- and continuous-valued functions and literals, much like in symbolic regression. The programs used in FEAT differ in two important ways. First, in contrast to typical SR, each individual n a set of such programs, the output of which is interpreted as a candidate representation, $n \rightarrow \phi(\mathbf{x}) = [\phi_1 \ldots \phi_m]$. Each program is constructed from a list of instructions is shown in Table 1. The second difference from traditional SR is that the weights of differentiable nodes are encoded in the edges between programs, as shown in Figure 1. The instructions include typical activation functions used in NN, e.g. tanh, sigmoid, logit and relu nodes, and the weights are encoded in a very similar manner. indeed, a fully connected feedforward NN is representable in FEAT's representation. However, due to the tree-based construction process and the use of elementary arithmetic operators (+,-,*,/), features in FEAT are biased to be thinly connected, and as a result, more legible.

2.3 Initialization

FEAT begins by fitting an ML model to the original data. For the examples in this paper, we use a linear model $\hat{y} = \mathbf{x}^T\beta$ trained using linear ridge regression. The values of β are used to set probabilities of sampling each predictor in $\mathbf{x}$ for use in a representation, according to Eqn. **??**. This initial representation, $\phi = \mathbf{x}$, is introduced into the original population, along with $P - 1$ randomly generated representations.

2.4 Variation

During variation, the representations are perturbed using a set of mutation and crossover methods. FEAT uses the following operators:

- Point mutation: changes a node type to one with matching output type and arity.
- Insert mutation: replaces a node with a randomly generated depth 1 subtree.
- Delete mutation: with equal probability, removes a feature or a replaces a sub-program with an input node.
- Insert/Delete dimension: adds or removes a new feature.
- Sub-tree crossover: replaces a sub-tree from one parent with the sub-tree of another parent.
- Dimension crossover: swaps two features between parents.

An example of these operations is shown in Figure **??**. The exact probabilities of each variation operator will affect the performance of the algorithm. For the purposes of our study, we use each operator with uniform probability.

Feedback The use of an ML model to assess the fitness of each representation can be used to provide information about the elements of the representation that should be changed. In particular, we assume that programs in the representation with small coefficients are the best candidates for mutation and crossover. With this in mind, we set the probability of mutation for each candidate program n with associated c coefficients β as:

$$\begin{aligned} \tilde{\beta}_i(n) &= |\beta_i| / \sum_i^c |\beta_i| \\ s_n &= \exp(1 - \tilde{\beta}_n) / \sum_i^c \exp(1 - \beta_i) \\ P_M(n) &= f s_n + (1 - f)\frac{1}{c} \end{aligned} \tag{4}$$

The term s_n in Eqn. 5 is the softmax-normalized inversions of the model coefficient norms $\tilde{\beta}$, which are between 0 and 1. Therefore the smaller the coefficient, the higher the probability of mutation. The parameter f is used to control the amount of feedback used to weight the probabilities; $\frac{1}{c}$ in this case represents uniform probability. Among nodes in program n, mutation occurs with uniform probability. An extension for differentiable nodes could be to weight this within-program probability by the magnitude of the weights associated with each node. However we expect this would yield diminishing returns.

Variation in symbolic regression typically consists of crossover and mutation. Most commonly the choice of which nodes to mutate or crossover is made uniform randomly. In the case of FEAT, we make use of the information provided by the ML model to weight the probabilities of mutation and crossover. The probability of a feature or its subtree being chosen for mutation is inversely proportional to the magnitude of its coefficient in the model, i.e. β in Eqn. **??**. To normalize the weights into probabilities, we use the softmax transformation, giving

$$p_{mutate}(\phi_i|\beta_i) = \exp(1 - |\beta_i|) / \sum_i^m \exp(1 - |\beta_i|) \tag{5}$$

2.5 Selection and Survival

The selection step selects among P parents those representations that will be used to generate offspring. Following variation, the population consists of $2P$ representations of parents and offspring. The survival step is used to reduce the population back to size P, at which point the generation is finished. We study five configurations of selection adopted from literature that are described below.

ϵ-lexicase selection This selection method (abbreviated Lex) was proposed for regression problems [10, 3] as an adaption of lexicase selection, a technique used primarily for discrete error problems. Under ϵ-lexicase selection, parents are chosen via population filtering using randomized orders of training samples with the ϵ threshold defined relative with respect to the sample loss among the pool. This filtering strategy scales probability of selection for an individual based on the difficulty of the training cases the individual performs well on. Lex has shown strong performance among symbolic regression methods in recent tests, motivating our interest in studying it [11].

Non-dominated sorting genetic algorithm 2 NSGA-2 is a popular selection and survival strategy for multi-objective optimization [**?**] that applies preference for selection based on Pareto dominance relations. One individual (n_i) is said to dominate another (n_j), i.e. $n_i n_j$, if, for all objectives, n_i performs at least as well as n_j, and for at least one objective, n_i strictly outperforms n_j. The Pareto *front* is the set of individuals in $\mathcal{N}$ that are non-dominated in the population. The Pareto front represents optimal trade-offs between objectives. We define two objectives in our study: the first corresponds to the squared loss function for individual n, and o_2 corresponds to the complexity of the representation. There are many ways to define complexity of an expression; one could simply look at the number of operations in a representation, or look at the behavioral complexity of the representation using a polynomial score. The one we use, which is similar to that used by others, is to assign a complexity weight to each operator (see Table 1), with higher weights assigned to operators considered more complex. If the weight of operator n is c_n, then the complexity of an expression tree beginning at node n is defined recursively as

$$C(n) = c_n * \sum_{a=1}^{k} C(a) \tag{6}$$

where n has k arguments, and $C(a)$ is the complexity of argument a. The complexity of a representation is then defined as the sum of the complexities of its output nodes. The goal of defining complexity in such a way is to discourage deep sub-expressions within complex nodes, which are often hard to interpret. It's important to note that the choice of operator weights is bound to be subjective, since we lack an objective notion of interpretability. For this reason, although we use Eqn. **??** to drive search, our experimental comparisons with other algorithms rely on the parameter counts of the final models for benchmarking interpretability of different methods.

NSGA-2 also relies on a behavioral diversity measure to measure the spread of solutions in objective space. Each individual is assigned a crowding distance measure, which is a meausure of its distance to its two adjacent neighbors in objective space.

Under NSGA-2, parent selection is conducted according to Pareto tournaments of size 2. In tournament selection, two parents are randomly drawn from the population and compared. If one dominates the other, it is chosen; crowding distnace is used to break ties.

The survival step of NSGA-2 begins by sorting the population according to their Pareto front *ranking*, which is a measure of their distance to the Pareto front. Individuals are added to the surviving population in order of their ranking. If a rank level does not completely fit, individuals of that rank are sorted by crowding distance and added in that order until P individuals are chosen for survival.

Simulated annealing Simulated annealing is a non-evolutionary technique that instead models the optimization process on the metallurgical process of annealing. In our implementation, offspring compete with their parents; in the case of multiple parents, offspring compete with the progam with which they share more nodes. The probability of an offspring replacing its parent in the population is given by the equation

$$P_{sel}(n_o|n_p, t) = \exp\left(\frac{F(n_p) - F(n_o)}{t}\right) \tag{7}$$

The probability of offspring replacing its parent is a function of its fitness, in our case the mean squared loss of the candidate model. In Eqn. 7, t is a scheduling parameter that controls the rate of "cooling", i.e. the rate at which steps in the search space that are worse are tolerated by the update rule. In accordance with [7], we use an exponential schedule for t, defined as $t_g = (0.9)^g t_0$, where g is the current generation and $t0$ is the starting temperature. $t0$ is set to 10 in our experiments.

Random search We compare the selection and survival methods to random search, in which no assumptions are made about the structure of the search space. To conduct random search, we randomly sample $\mathbb{S}$ using the initialization procedure described above. Since FEAT begins with a linear model of the process, random search will produce a representation at least as good as this initial model on the internal validation set.

2.6 Archiving

During optimization, FEAT maintains a separate population that acts as an archive. The archive maintains a Pareto front according to minimum loss and complexity (Eqn **??**). At the end of optimization, the archive is tested on a small hold-out validation set. The indivdiual with the lowest validation loss is the final selected model. Maintaining this archive helps protect against overfitting resulting from overly complex / high capacity representations, and also can be interepreted directly to help understand the process being modelled.

3 Related Work

FEAT is primarily motivated by symbolic regression approaches to feature engineering[9**? ?**] that use GP to search for possible representations and couple with an ML model to handle the parametrization of the representations. FEAT differs from these methods in the following ways. A key challenge in symbolic regression is understanding functional modularity within representations/programs that can be exploited for search. FEAT is designed with the insight that ML weights can be leveraged during variation to promote functional building blocks, an exploit not used in previous methods. Second, FEAT uses multiple type representations, and thus can learn continuos and rule-based features within a single representation, unlike previous methods. This is made possible using a stack-based encoding with strongly-typed operators. Finally, FEAT incorporates two elements of NN learning to improve its representational capacity: activation functions commonly used in NN and edge-based encoding of weights. Traditionally, SR operates with standard mathematical operators, and treats constants as leaves in the expression trees rather than edge weights. An exception is MRGP [**?**], which encodes weights at each node but updates them via Lasso instead of using gradient descent with backpropagation.

SR methods have also been paired with various parameter learning strategies, including those based on backpropagation [14, 8, 5]. Still, several systems using stochastic hill climbing have found continued use. The trade-off between time spent training weights versus searching among candidate model structures must be appreciated. It is not entirely clear how one should learn weights for nodes that evaluated in several different environments via their promulgation into new programs and varying locations.

The idea to evolve NN architectures is well established in literature, and is known as neuroevolution. Popular methods of neuroevolution include neuroevolution of augmenting topologies (NEAT and Hyper-NEAT), compositional pattern producing networks (CPPN). Traditionally these approaches eschew the parameter learing step common in other NN paradigms, although others have developed integrations. These methods do not have interepretability as a core focus, and thus do not attempt to use multi-objective methods to update the networks.

Dropout FEAT also bears motivational relationship to dropout, a popular method of NN regularization. Dropout is an approach that may improve interpretability of models by considering competing subsets of networks during training[13]. The authors were motivated in part by the evolutionary process of sexual recombination, which is a dominant form of genotype variation found in nature for unclear reasons. One reason the authors entertain is the ability of crossover between models to assert selective pressure for genes to be robust to different environmental contexts, since crossover may introduce large changes to neighboring genes. This pressure is also rewards genes with modular functionality since genes that are close together are more likely to be shared together and must perform a similar function in a new organism. Experimental investigations have found that changing environmental contexts are important for the development of network modularity [6] in natural systems.

Symbolic Regression Symbolic regression of dynamical systems[12]

Table 2: Configurations tested for FEAT. Multiple values indicate use in hyperparameter tuning.

Setting	Values
SO Method	NSGA2, Lex, Lex-NSGA2, SimAnn, Random
Population size	100
Generations	100
Max depth	10
Max dimensionality	50
Fitness	R2
Parameter learning	{hillclimbing, SGD}
Learning rate	0.1
Iterations / individual / generation	{1,10,100}
Crossover rate	0.5

Table 3: Neural Network configurations.

Setting	Sklearn-MLP	Torch-NN
Optimizer	Adam	SGD
Hidden Layers	{1,3,6}	6
Neurons per layer	10	10
Learning rate	(initial) {1e-3, 1e-2, 1e-1}	{1e-3, 1e-2, 1e-1}
Regularization	L_2, α = {1e-4, 1e-2, 1e-1}	Dropout, p=0.5
Iterations	2000	{500, 1000 }
Early Stopping	True	False

Global symbolic regression[1]

4 Experiment

Real world classification and regression

5 Results

6 Discussion and Conclusion

References

References follow the acknowledgments. Use unnumbered first-level heading for the references. Any choice of citation style is acceptable as long as you are consistent. It is permissible to reduce the font size to `small` (9 point) when listing the references. **Remember that you can use more than eight pages as long as the additional pages contain *only* cited references.**

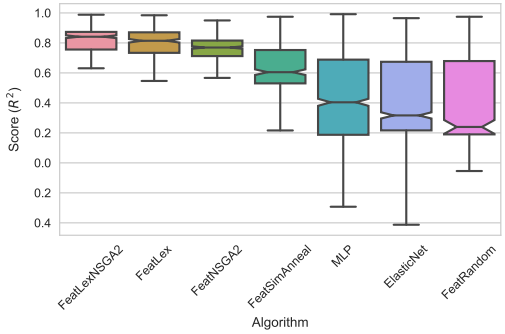

Figure 2: Mean 10-fold CV R^2 performance for various SO methods in comparison to other ML methods, across the benchmark problems.

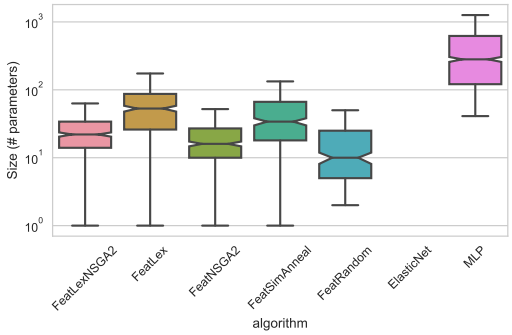

Figure 3: Size comparisons of the final models in terms of number of parameters.

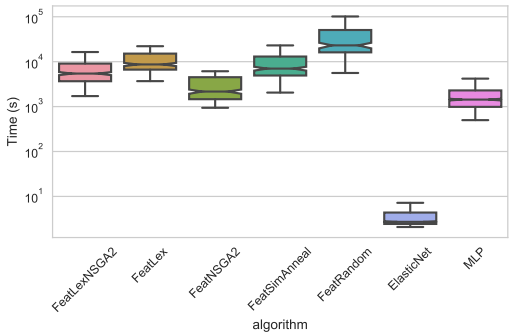

Figure 4: Wall-clock runtime for each method in seconds.

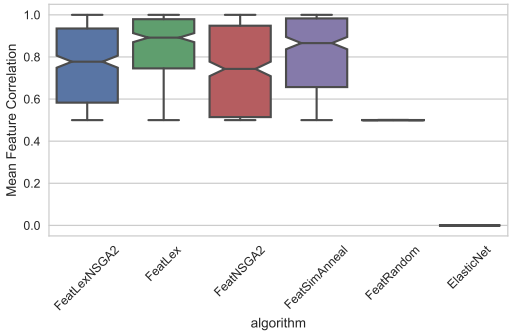

Figure 5: Mean correlation between engineered features for different SO methods compared to the correlations in the original data (ElasticNet).

References

[1] Austel, V., Dash, S., Gunluk, O., Horesh, L., Liberti, L., Nannicini, G., and Schieber, B. (2017). Globally Optimal Symbolic Regression. *arXiv:1710.10720 [stat]*. arXiv: 1710.10720.

[2] Bengio, Y., Courville, A., and Vincent, P. (2013). Representation learning: A review and new perspectives. *IEEE transactions on pattern analysis and machine intelligence*, 35(8):1798–1828.

[3] Cava, W. L., Helmuth, T., Spector, L., and Moore, J. H. (2018). A probabilistic and multi-objective analysis of lexicase selection and -lexicase selection. *Evolutionary Computation*, pages 1–28.

[4] Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep Learning*. MIT Press.

[5] Izzo, D., Biscani, F., and Mereta, A. (2017). Differentiable Genetic Programming. In *European Conference on Genetic Programming*, pages 35–51. Springer.

[6] Kashtan, N. and Alon, U. (2005). Spontaneous evolution of modularity and network motifs. *Proceedings of the National Academy of Sciences*, 102(39):13773–13778.

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

[8] Kommenda, M., Kronberger, G., Winkler, S., Affenzeller, M., and Wagner, S. (2013). Effects of constant optimization by nonlinear least squares minimization in symbolic regression. In Blum, C., Alba, E., Bartz-Beielstein, T., Loiacono, D., Luna, F., Mehnen, J., Ochoa, G., Preuss, M., Tantar, E., and Vanneschi, L., editors, *GECCO '13 Companion: Proceeding of the fifteenth annual conference companion on Genetic and evolutionary computation conference companion*, pages 1121–1128, Amsterdam, The Netherlands. ACM.

[9] La Cava, W. and Moore, J. (2017). A General Feature Engineering Wrapper for Machine Learning Using \epsilon -Lexicase Survival. In *Genetic Programming*, pages 80–95. Springer, Cham.

[10] La Cava, W., Spector, L., and Danai, K. (2016). Epsilon-Lexicase Selection for Regression. In *Proceedings of the Genetic and Evolutionary Computation Conference 2016*, GECCO '16, pages 741–748, New York, NY, USA. ACM.

[11] Orzechowski, P., La Cava, W., and Moore, J. H. (2018). Where are we now? A large benchmark study of recent symbolic regression methods. *arXiv:1804.09331 [cs]*. arXiv: 1804.09331.

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

[13] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2014). Dropout: A simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958.

[14] Topchy, A. and Punch, W. F. (2001). Faster genetic programming based on local gradient search of numeric leaf values. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)*, pages 155–162.