DEEP SYMBOLIC REGRESSION: RECOVERING MATHEMATICAL EXPRESSIONS FROM DATA VIA RISK-SEEKING POLICY GRADIENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Discovering the underlying mathematical expressions describing a dataset is a core challenge for artificial intelligence. This is the problem of *symbolic regression*. Despite recent advances in training neural networks to solve complex tasks, deep learning approaches to symbolic regression are underexplored. We propose a framework that leverages deep learning for symbolic regression via a simple idea: use a large model to search the space of small models. Specifically, we use a recurrent neural network to emit a distribution over tractable mathematical expressions and employ a novel risk-seeking policy gradient to train the network to generate better-fitting expressions. Our algorithm outperforms several baseline methods (including Eureqa, the gold standard for symbolic regression) in its ability to exactly recover symbolic expressions on a series of benchmark problems, both with and without added noise. More broadly, our contributions include a framework that can be applied to optimize hierarchical, variable-length objects under a black-box performance metric, with the ability to incorporate constraints in situ, and a risk-seeking policy gradient formulation that optimizes for best-case performance instead of expected performance.

1 INTRODUCTION

Understanding the mathematical relationships among variables in a physical system is an integral component of the scientific process. Symbolic regression aims to identify these relationships by searching over the space of tractable (i.e. concise, closed-form) mathematical expressions to best fit a dataset. Specifically, given a dataset (X, y), where each point $X_i \in \mathbb{R}^n$ and $y_i \in \mathbb{R}$, symbolic regression aims to identify a function $f : \mathbb{R}^n \to \mathbb{R}$ that best fits the dataset, where the functional form of f is a short mathematical expression. The resulting expression can be readily interpreted and/or provide useful scientific insights simply by inspection. In contrast, conventional regression imposes a single model structure that is fixed during training, often chosen to be expressive (e.g. a neural network) at the expense of being easily interpretable.

Symbolic regression exhibits several unique features that make it an excellent test problem for benchmarking automated machine learning (AutoML) and program synthesis methods: (1) there exist well-established, challenging benchmark problems with stringent success criteria (White et al., 2013); (2) there exist well-established baseline methods (most notably, the Eureqa algorithm (Schmidt & Lipson, 2009)); and (3) the reward function is computationally expedient, allowing sufficient experiment replicates to achieve statistical significance. Most other AutoML tasks, e.g. neural architecture search (NAS), do not exhibit these features; in fact, even simply *evaluating* the efficiency of the discrete search itself is a known challenge within NAS (Yu et al., 2019).

The space of mathematical expressions is discrete (in model structure) and continuous (in model parameters), growing exponentially with the length of the expression, rendering symbolic regression a challenging machine learning problem—thought to be NP-hard (Lu et al., 2016). Given this large, combinatorial search space, traditional approaches to symbolic regression typically utilize evolutionary algorithms, especially genetic programming (GP) (Koza, 1992; Schmidt & Lipson, 2009; Bäck et al., 2018). In GP-based symbolic regression, a population of mathematical expressions is "evolved" using evolutionary operations like selection, crossover, and mutation to improve a fitness

function. While GP can be effective, it is also known to scale poorly to larger problems and to exhibit high sensitivity to hyperparameters.

Deep learning has permeated almost all areas of artificial intelligence, from computer vision (Krizhevsky et al., 2012) to optimal control (Mnih et al., 2015). However, deep learning may seem incongruous with or even antithetical toward symbolic regression, given that neural networks are typically highly complex, difficult to interpret, and rely on gradient information. We propose a framework that resolves this incongruity by tying deep learning and symbolic regression together with a simple idea: use a large model (i.e. neural network) to search the space of small models (i.e. symbolic expressions). This framework leverages the representational capacity of neural networks to generate interpretable expressions, while entirely bypassing the need to interpret the network itself.

We present *deep symbolic regression* (DSR), a gradient-based approach for symbolic regression based on reinforcement learning. In DSR, a recurrent neural network (RNN) emits a distribution over mathematical expressions. Expressions are sampled from the distribution, instantiated, and evaluated based on their fitness to the dataset. This fitness is used as the reward signal to train the RNN using a novel risk-seeking policy gradient algorithm. As training proceeds, the RNN adjusts the likelihood of an expression relative to its reward, assigning higher probabilities to better expressions.

We demonstrate that DSR outperforms several baseline methods, including two commercial software algorithms. We summarize our contributions as follows: (1) a novel method for symbolic regression that outperforms several baselines on a set of benchmark problems, (2) an autoregressive generative modeling framework for optimizing hierarchical, variable-length objects that accommodates in situ constraints, and (3) a novel risk-seeking policy gradient objective and accompanying Monte Carlo estimation procedure that optimizes for best-case performance instead of average performance.

2 Related Work

Deep learning for symbolic regression. Several recent approaches leverage deep learning for symbolic regression. AI Feynman (Udrescu & Tegmark, 2020) propose a problem-simplification tool for symbolic regression. They use neural networks to identify simplifying properties in a dataset (e.g. multiplicative separability, translational symmetry), which they exploit to recursively define simplified sub-problems that can then be tackled using any symbolic regression algorithm. In GrammarVAE, Kusner et al. (2017) develop a generative model for discrete objects that adhere to a pre-specified grammar, then optimize them in latent space. They demonstrate this can be used for symbolic regression; however, the method struggles to exactly recover expressions, and the generated expressions are not always syntactically valid. Sahoo et al. (2018) develop a symbolic regression framework using neural networks whose activation functions are symbolic operators. While this approach enables an end-to-end differentiable system, backpropagation through activation functions like division or logarithm requires the authors to make several simplifications to the search space, ultimately precluding learning certain simple classes of expressions like $\sqrt{x}$ or $\sin(x/y)$. We address and/or directly compare to these works in Appendices C and E.

AutoML and program synthesis. Symbolic regression is related to both automated machine learning (AutoML) and program synthesis, in that they all involve a search for an executable program (i.e. expression) to solve a particular task (i.e. to fit data) (Abolafia et al., 2018; Devlin et al., 2017; Riedel et al., 2016). More specifically, our framework has many parallels to a body of works within AutoML that use an autoregressive RNN to define a distribution over discrete objects and use reinforcement learning to optimize this distribution under a black-box performance metric (Zoph & Le, 2017; Ramachandran et al., 2017; Bello et al., 2017; Abolafia et al., 2018). For example, in neural architecture search (Zoph & Le, 2017), an RNN searches the space of neural network architectures, encoded by a sequence of discrete "tokens" specifying architectural properties (e.g. number of neurons) of each layer. The length of the sequence is fixed or scheduled during training; in contrast, our framework defines a search space that is both inherently hierarchical and variable length. Ramachandran et al. (2017) search the space of neural network activation functions. While this space is hierarchical in nature, the authors (rightfully) constrain it substantially by positing a functional unit that is repeated sequentially, thus restricting their search space back to a fixed-length sequence. However, a repeating-unit constraint is not practical for symbolic regression because the ground truth expression may have arbitrary structure.

Autoregressive models. The RNN-based distribution over expressions used in DSR is autoregressive, meaning each token is conditioned on the previously sampled tokens. Autoregressive models have proven to be useful for audio and image data (Oord et al., 2016a;b) in addition to the AutoML works discussed above; we further demonstrate their efficacy for hierarchical expressions. GraphRNN defines a distribution over graphs that generates an adjacency matrix one column at a time in autoregressive fashion (You et al., 2018). In principle, GraphRNN could be constrained to define a distribution over expressions, since trees are a special case of graphs. However, GraphRNN constructs graphs breadth-first, whereas expressions are more naturally represented using depth-first traversals (Li et al., 2005). Further, DSR exploits the hierarchical nature of trees by providing the parent and sibling as inputs to the RNN, and leverages the additional structure of expression trees that a node's value determines its number of children (e.g. cosine is a unary operator and thus has one child).

Risk-aware reinforcement learning. Many of the AutoML methods discussed above suffer from what we call the "expectation problem." That is, policy gradient methods are fundamentally suited for optimizing *expectations*; however, domains like neural architecture search and symbolic regression are evaluated by the few or single best-performing samples. Thus, there is a mismatch between the training objective function and the true desired objective: to maximize best-case performance. Abolafia et al. (2018) address the expectation problem by maintaining a priority queue of the best seen samples and using supervised learning to increase the likelihood of those top samples. Similarly, Liang et al. (2018) use a memory buffer to augment a policy gradient with off-policy training. These methods, however, only apply in the context of reinforcement learning environments with both deterministic transition dynamics and deterministic rewards. In contrast, the *risk-seeking policy gradient* introduced here is general, applying to any reinforcement learning environment and any stochastic policy gradient algorithm trained using batches, e.g. on Atari using proximal policy optimization (Schulman et al., 2017). Lastly, our risk-seeking policy gradient is closely related to the EPOpt-ε algorithm used for robust reinforcement learning (Rajeswaran et al., 2016), which is based on a risk-averse policy gradient formulation (Tamar et al., 2014).

3 METHODS

Our overall approach involves representing mathematical expressions as sequences, developing an autoregressive model to generate expressions under a pre-specified set of constraints, and developing a risk-seeking policy gradient to train the model to generate better-fitting expressions.

3.1 GENERATING EXPRESSIONS WITH A RECURRENT NEURAL NETWORK

We leverage the fact that mathematical expressions can be represented using *symbolic expression trees*. Expression trees are a type of binary tree in which internal nodes are mathematical operators and terminal nodes are input variables or constants. Operators may be unary (i.e. one argument, such as sine) or binary (i.e. two arguments, such as multiply). Further, we can represent an expression tree as a sequence of node values or "tokens" by using its pre-order traversal (i.e. by visiting each node depth-first, then left-to-right). This allows us to generate an expression tree sequentially while still maintaining a one-to-one correspondence between the tree and its traversal. Thus, we represent an expression τ by the pre-order traversal of its corresponding expression tree. We denote the ith token of the traversal as τ_i and the length of the traversal as $|\tau| = T$. Each token has a value within a given library $\mathcal{L}$ of possible tokens, e.g. $\{+, -, \times, \div, \sin, x\}$.

We generate expressions one token at a time along the pre-order traversal (from τ_1 to τ_T). A categorical distribution with parameters ψ defines the probabilities of selecting each token from $\mathcal{L}$. To capture the "context" of the expression as it is being generated, we condition this probability upon the selections of all previous tokens in that traversal. This conditional dependence can be achieved very generally using an RNN with parameters θ that emits a probability vector ψ in an autoregressive manner. Specifically, the ith output of the RNN passes through a softmax layer (with shared weights across time steps) to produce vector $\psi^{(i)}$, which defines the probability distribution for selecting the ith token τ_i, conditioned on the previously selected tokens $\tau_{1:(i-1)}$. That is, $p(\tau_i|\tau_{1:(i-1)};\theta) = \psi^{(i)}_{\mathcal{L}(\tau_i)}$, where $\mathcal{L}(\tau_i)$ is the index in $\mathcal{L}$ corresponding to τ_i. The likelihood of the entire sampled expression is simply the product of the likelihoods of its tokens: $p(\tau|\theta) = \prod_{i=1}^{|\tau|} p(\tau_i|\tau_{1:(i-1)};\theta) = \prod_{i=1}^{|\tau|} \psi^{(i)}_{\mathcal{L}(\tau_i)}$.

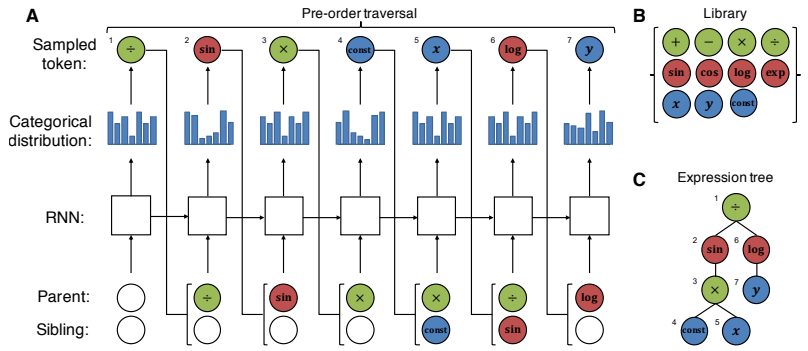

Figure 1: **A**. Example of sampling an expression from the RNN. For each token, the RNN emits a categorical distribution over tokens, a token is sampled, and the parent and sibling of the next token are used as the next input to the RNN. Subsequent tokens are sampled autoregressively until the tree is complete (i.e. all tree branches reach terminal nodes). The resulting sequence of tokens is the tree's pre-order traversal, which can be used to reconstruct the tree and instantiate its corresponding expression. Colors correspond to the number of children for each token. White circles represent empty tokens. **B**. The library of tokens. **C**. The expression tree sampled in **A**.

An example of the sampling process is illustrated in Figure 1; pseudocode is provided in Algorithm 2 in Appendix A. Note that different samples from the distribution have different tree structures of different size; thus, the search space is inherently both hierarchical and variable length.

Providing hierarchical inputs to the RNN. Conventionally, the input to the RNN when sampling a token would be a representation of the previously sampled token. However, the search space for symbolic regression is inherently hierarchical, and the previously sampled token may actually be very distant from the next token to be sampled in the expression tree. For example, the fifth and sixth tokens sampled in Figure 1 are adjacent nodes in the traversal but are four edges apart in the expression tree. To better capture hierarchical information, we provide as inputs to the RNN a representation of the parent and sibling nodes of the token being sampled. We introduce an empty token for cases in which a node does not have a parent or sibling. Pseudocode for identifying the parent and sibling nodes given a partial traversal is provided in Subroutine 1 in Appendix A.

Constraining the search space. Under our framework, it is straightforward to apply a priori constraints to reduce the search space. To demonstrate, we impose several simple, domain-agnostic constraints: (1) Expressions are limited to a pre-specified minimum and maximum length. We selected minimum length of 4 to prevent trivial expressions and a maximum length of 30 to ensure expressions are tractable. (2) The children of an operator should not all be constants, as the result would simply be a different constant. (3) The child of a unary operator should not be the inverse of that operator, e.g. $\log(\exp(x))$ is not allowed. (4) Descendants of trigonometric operators should not be trigonometric operators, e.g. $\sin(x + \cos(x))$ is not allowed. While still semantically meaningful, such composed trigonometric operators do not appear in virtually any scientific domain.

We apply these constraints in situ (i.e. concurrently with autoregressive sampling) by zeroing out the probabilities of selecting tokens that would violate a constraint. Pseudocode for this process is provided in Subroutine 2 in Appendix A. This process ensures that samples always adhere to all constraints, without rejecting samples post hoc. In contrast, imposing constraints with GP requires rejecting evolutionary operations post hoc (Fortin et al., 2012), which can be problematic (Craenen et al., 2001), and as we show in our experiments, can actually reduce performance.

Reward function. Once a pre-order traversal is sampled, we instantiate the corresponding symbolic expression and evaluate it with a reward function. A standard fitness measure in GP-based symbolic regression is normalized root-mean-square error (NRMSE), the root-mean-square error normalized by the standard deviation of the target values, σ_y. That is, given a dataset (X, y) of size n and candidate expression f, NRMSE $= \frac{1}{\sigma_y}\sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - f(X_i))^2}$. To bound the reward function, we apply a squashing function: $R(\tau) = 1/(1 + \text{NRMSE})$.

Constant optimization. If the library $\mathcal{L}$ includes the constant token, sampled expressions may include several constant placeholders. These can be viewed as parameters ξ of the symbolic expression, which we optimize by maximizing the reward function: $\xi^\star = \arg\max_\xi R(\tau;\xi)$, using a nonlinear optimization algorithm, e.g. BFGS (Fletcher, 2013). We perform this inner optimization loop for each sampled expression as part of the reward computation before performing each training step.

3.2 Training the RNN using policy gradients

Standard policy gradient. Now that we have a distribution over mathematical expressions $p(\tau|\theta)$, we first consider the standard policy gradient objective to maximize $J_{\text{std}}(\theta)$, defined as the expectation of a reward function $R(\tau)$ under expressions from the distribution: $J_{\text{std}}(\theta) \doteq \mathbb{E}_{\tau\sim p(\tau|\theta)}\left[R(\tau)\right]$. The standard REINFORCE policy gradient (Williams, 1992) can be used to maximize this expectation via gradient ascent:

$$\begin{aligned}\nabla_\theta J_{\text{std}}(\theta) &= \nabla_\theta \mathbb{E}_{\tau\sim p(\tau|\theta)}\left[R(\tau)\right] \\ &= \mathbb{E}_{\tau\sim p(\tau|\theta)}\left[R(\tau)\nabla_\theta \log p(\tau|\theta)\right]\end{aligned}$$

This result allows one to estimate the expectation using samples from the distribution. Specifically, an unbiased estimate of $\nabla_\theta J_{\text{std}}(\theta)$ can be obtained by computing the sample mean over a batch of N sampled expressions $\mathcal{T} = \{\tau^{(i)}\}_{i=1}^N$:

$$\nabla_\theta J_{\text{std}}(\theta) \approx \frac{1}{N}\sum_{i=1}^{N} R(\tau^{(i)})\nabla_\theta \log p(\tau^{(i)}|\theta)$$

This is an unbiased gradient estimate, but in practice it has high variance. To reduce variance, it is common to subtract a baseline function b from the reward. As long as the baseline is not a function of the current batch of expressions, the gradient estimate is still unbiased. Common choices of baseline functions are a moving average of rewards or an estimate of the value function.

Risk-seeking policy gradient. The standard policy gradient objective, $J_{\text{std}}(\theta)$, is defined as an *expectation*. This is the desired objective for control problems in which one seeks to optimize the average performance of a policy. However, in domains like symbolic regression, program synthesis, or neural architecture search, the final performance is measured by the single or few best-performing samples found during training. Similarly, one might be interested in a policy that achieves a "high score" in a control environment (e.g. Atari). For such problems, $J_{\text{std}}(\theta)$ is not an appropriate objective, as there is a mismatch between the objective being optimized and the final performance metric; this is the "expectation problem." To address this disconnect, we propose an alternative objective that focuses learning only on *maximizing best-case performance*. We first define $R_\varepsilon(\theta)$ as the $(1-\varepsilon)$-quantile of the distribution of rewards under the current policy. We then propose a new learning objective, $J_{\text{risk}}(\theta;\varepsilon)$, parameterized by ε:

$$J_{\text{risk}}(\theta;\varepsilon) \doteq \mathbb{E}_{\tau\sim p(\tau|\theta)}\left[R(\tau) \mid R(\tau) \geq R_\varepsilon(\theta)\right] \tag{1}$$

This objective aims to increase the reward of the top ε fraction of samples from the distribution, without regard for samples below that threshold. This objective bears close resemblance with ε-conditional value at risk (CVaR), for which the "$\leq$" symbol is used instead of "$\geq$" and the ε-quantile of rewards is used instead of the $(1-\varepsilon)$-quantile. Optimizing CVaR is a form of risk-averse learning that results in a policy that is robust against catastrophic outcomes (Tamar et al., 2014; Rajeswaran et al., 2016). In contrast, optimizing $J_{\text{risk}}(\theta;\varepsilon)$ yields a *risk-seeking* policy gradient that aims to increase best-case performance at the expense of lower worst-case and average performances. Next, we show the analogous policy gradient of $J_{\text{risk}}(\theta;\varepsilon)$ and how to estimate it via Monte Carlo sampling.

Proposition 1. *Let $J_{risk}(\theta;\varepsilon)$ denote the conditional expectation of rewards above the $(1-\varepsilon)$-quantile, as in Equation (1). Then the gradient of $J_{risk}(\theta;\varepsilon)$ is given by:*

$$\nabla_\theta J_{risk}(\theta;\varepsilon) = \mathbb{E}_{\tau\sim p(\tau|\theta)}[(R(\tau) - R_\varepsilon(\theta)) \cdot \nabla_\theta \log p(\tau|\theta) \mid R(\tau) \geq R_\varepsilon(\theta)]$$

The proof and assumptions are provided in Appendix B, and are adapted from the policy gradient derivation for the CVaR objective (Tamar et al., 2014). This result suggests a simple Monte Carlo estimate of the gradient from a batch of N samples:

$$\nabla_\theta J_{\text{risk}}(\theta;\varepsilon) \approx \frac{1}{\varepsilon N}\sum_{i=1}^{N}\left[R(\tau^{(i)}) - \tilde{R}_\varepsilon(\theta)\right] \cdot \mathbf{1}_{R(\tau^{(i)})\geq \tilde{R}_\varepsilon(\theta)}\nabla_\theta \log p(\tau^{(i)}|\theta),$$

Algorithm 1 Deep symbolic regression with risk-seeking policy gradient

input learning rate α; entropy coefficient $\lambda_{\mathcal{H}}$; risk factor ε; batch size N; reward function R
output Best fitting expression $\tau^\star$

Initialize RNN with parameters θ, defining distribution over expressions $p(\cdot|\theta)$
repeat
  $\mathcal{T} \leftarrow \{\tau^{(i)} \sim p(\cdot|\theta)\}_{i=1}^N$ ▷ Sample N expressions (Alg. 2 in Appendix A)
  $\mathcal{T} \leftarrow \{\text{OptimizeConstants}(\tau^{(i)}, R)\}_{i=1}^N$ ▷ Optimize constants w.r.t. reward function
  $\mathcal{R} \leftarrow \{R(\tau^{(i)})\}_{i=1}^N$ ▷ Compute rewards
  $R_\varepsilon \leftarrow (1-\varepsilon)$-quantile of $\mathcal{R}$ ▷ Compute reward threshold
  $\mathcal{T} \leftarrow \{\tau^{(i)} : R(\tau^{(i)}) \geq R_\varepsilon\}$ ▷ Select subset of expressions above threshold
  $\mathcal{R} \leftarrow \{R(\tau^{(i)}) : R(\tau^{(i)}) \geq R_\varepsilon\}$ ▷ Select corresponding subset of rewards
  $\hat{g}_1 \leftarrow \text{ReduceMean}((\mathcal{R} - R_\varepsilon)\nabla_\theta \log p(\mathcal{T}|\theta))$ ▷ Compute risk-seeking policy gradient
  $\hat{g}_2 \leftarrow \text{ReduceMean}(-\lambda_{\mathcal{H}} \nabla_\theta \mathcal{H}(\mathcal{T}|\theta))$ ▷ Compute entropy gradient
  $\theta \leftarrow \theta + \alpha(\hat{g}_1 + \hat{g}_2)$ ▷ Apply gradients
  if $\max \mathcal{R} > R(\tau^\star)$ **then** $\tau^\star \leftarrow \tau^{(\arg\max \mathcal{R})}$ ▷ Update best expression
return $\tau^\star$

where $\tilde{R}_\varepsilon(\theta)$ is the empirical $(1-\varepsilon)$-quantile of the batch of rewards, and $\mathbf{1}_x$ returns 1 if condition x is true and 0 otherwise. Essentially, this is the standard REINFORCE Monte Carlo estimate with two differences: (1) theory suggests a specific baseline, $R_\varepsilon(\theta)$, whereas the baseline for standard policy gradients is non-specific, chosen by the user; and (2) effectively, only the top ε fraction of samples from each batch are used in the gradient computation. This process is essentially the opposite of the approach used to optimize CVaR (Tamar et al., 2014; Rajeswaran et al., 2016) for risk-averse reinforcement learning, in which only the *bottom* ε fraction of samples from each batch are used.

Lastly, in accordance with the maximum entropy reinforcement learning framework (Haarnoja et al., 2018), we provide a bonus to the loss function proportional to the entropy of the sampled expressions. Pseudocode for DSR is shown in Algorithm 1. Source code is made available at {*anonymous-URL*}.

4 Results and Discussion

Evaluating DSR. We evaluated DSR on the Nguyen symbolic regression benchmark suite (Uy et al., 2011), a set of 12 commonly used benchmark expressions developed and vetted by the symbolic regression community (White et al., 2013). Each benchmark is defined by a ground truth expression, a training and test dataset, and a set of allowed operators, described in Table 2 in Appendix D. The training data is used to compute the reward for each candidate expression, the test data is used to evaluate the best found candidate expression at the end of training, and the ground truth expression is used to determine whether the best found candidate expression was correctly recovered.

We compared DSR against five strong symbolic regression baselines: (1) **PQT**: an implementation of our framework trained using priority queue training (Abolafia et al., 2018) in place of the risk-seeking policy gradient, (2) **VPG**: a "vanilla" implementation of our framework using the standard policy gradient (with baseline) in place of the risk-seeking policy gradient, (3) **GP**: a standard GP-based symbolic regression implementation (Fortin et al., 2012), (4) **Eureqa**: popular commercial software[1] based on Schmidt & Lipson (2009), the gold standard for symbolic regression, and (5) **Wolfram**: commercial software[2] based on Markov chain Monte Carlo and nonlinear regression (Fortuna, 2015). Each baseline is further detailed in Appendix C. Notably, the two RNN-based baselines (PQT and VPG) differ from DSR only via their training objective; all other aspects of our framework (in situ constraints, hierarchical RNN inputs, constant optimization) are also included.

Each experiment consists of 2 million expression evaluations for DSR, PQT, VPG, and GP, by which point training curves have levelled off. Hyperparameters were tuned on benchmarks Nguyen-7 and Nguyen-10 using a grid search comprising 800 hyperparameter combinations for GP and 81 combinations for each of DSR, PQT, and VPG (see Appendix D for details). Commercial software algorithms (Eureqa and Wolfram) do not expose hyperparameters and were run until completion.

[1]Sold by DataRobot, Inc. (www.datarobot.com) and formerly by Nutonian, Inc. (www.nutonian.com).
[2]Sold by Wolfram Research, Inc. as a part of Mathematica (www.wolfram.com/mathematica).

Table 1: Recovery rate comparison of DSR and five baselines on the Nguyen symbolic regression benchmark suite. A bold value represents statistical significance ($p < 10^{-3}$) across all benchmarks.

Benchmark	Expression	DSR	PQT	VPG	GP	Eureqa	Wolfram
Nguyen-1	$x^3 + x^2 + x$	100%	100%	96%	100%	100%	100%
Nguyen-2	$x^4 + x^3 + x^2 + x$	100%	99%	47%	97%	100%	100%
Nguyen-3	$x^5 + x^4 + x^3 + x^2 + x$	100%	86%	4%	100%	95%	100%
Nguyen-4	$x^6 + x^5 + x^4 + x^3 + x^2 + x$	100%	93%	1%	100%	70%	100%
Nguyen-5	$\sin(x^2)\cos(x) - 1$	72%	73%	5%	45%	73%	2%
Nguyen-6	$\sin(x) + \sin(x + x^2)$	100%	98%	100%	91%	100%	1%
Nguyen-7	$\log(x + 1) + \log(x^2 + 1)$	35%	41%	3%	0%	85%	0%
Nguyen-8	$\sqrt{x}$	96%	21%	5%	5%	0%	71%
Nguyen-9	$\sin(x) + \sin(y^2)$	100%	100%	100%	100%	100%	–
Nguyen-10	$2\sin(x)\cos(y)$	100%	91%	99%	76%	64%	–
Nguyen-11	x^y	100%	100%	100%	7%	100%	–
Nguyen-12	$x^4 - x^3 + \frac{1}{2}y^2 - y$	0%	0%	0%	0%	0%	–
	Average	**83.6%**	75.2%	46.7%	60.1%	73.9%	–

All experiments were replicated with 100 different random seeds for each benchmark expression. Wolfram results are only presented for one-dimensional benchmarks because the method is not applicable to higher dimensions. Additional experiment details are provided in Appendix D. Training curves are provided in Appendix F.

In Table 1, we report the recovery rate for each benchmark. We use the strictest definition of recovery: exact symbolic equivalence, as determined using a computer algebra system, e.g. SymPy (Meurer et al., 2017). In Table 9 in Appendix F, we report recovery on several additional variants of Nguyen benchmarks in which we introduced real-valued constants (to demonstrate the constant optimizer) or altered the functional form to make the problems more challenging. DSR significantly outperforms all five baselines in its ability to exactly recover benchmark expressions. In Tables 5–8 in Appendix E, we compare DSR to literature-reported values from four additional baseline methods (Huynh et al., 2016; Kusner et al., 2017; Jin et al., 2019; Trujillo et al., 2016) by carefully recapitulating their experimental setup. DSR greatly outperforms each study's published results.

Characterizing the risk-seeking policy gradient. The intuition behind the risk-seeking policy gradient is that it explicitly optimizes for best-case performance, possibly at the expense of average performance. We demonstrate this visually in Figure 2 by comparing the empirical reward distributions when trained with either the risk-seeking or standard policy gradient for Nguyen-8. (Analogous plots for all Nguyen benchmarks are provided in Appendix F.) Interestingly, at the end of training, the mean reward over the full batch (an estimate of $J_{\text{std}}(\theta)$) is larger when training with the standard policy gradient, even though the risk-seeking policy gradient produces larger mean over the top ε fraction of the batch (an estimate of $J_{\text{risk}}(\theta; \varepsilon)$) and a superior best expression. This is consistent with the intuition of maximizing best-case performance at the expense of average performance. In contrast, the best-case performance of the standard policy gradient plateaus early in training (Figure 2E, dashed blue curve), whereas the risk-seeking policy gradient continues to increase until the end of training (Figure 2E, dashed black curve).

Ablation studies. Algorithm 1 includes several additional components relative to a "vanilla" policy gradient search. We performed a series of ablation studies to quantify the effect of each of these components, along with the effects of the various constraints on the search space. In Figure 3, we show recovery rate for DSR on the Nguyen benchmarks for each ablation. While no single ablation leads to catastrophic failure, combinations of ablations can cause large degradation in performance.

Noisy data and amount of data. We evaluated the robustness of DSR to noisy data by adding independent Gaussian noise to the dependent variable, with mean zero and standard deviation proportional to the root-mean-square of the dependent variable in the training data. In Figure 4, we varied the proportionality constant from 0 (noiseless) to 10^{-1} and evaluated each algorithm (except Wolfram, which catastrophically fails for even the smallest noise level) across all Nguyen benchmarks. Because expressions can overfit to the noise in these experiments, we defined recovery as exact symbolic equivalence on any expression along the reward-complexity Pareto front at the end of training (see Appendix D for details). Increasing the dataset size may help prevent overfitting by

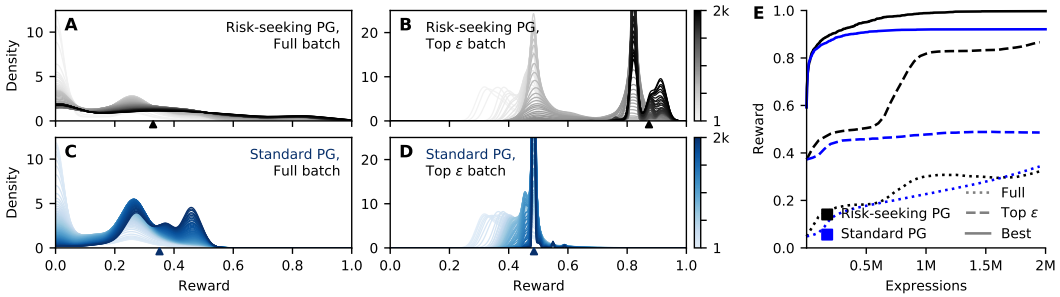

Figure 2: **A** - **D**. Empirical reward distributions for Nguyen-8. Each curve is a Gaussian kernel density estimate (bandwidth 0.25) of the rewards for a particular training iteration, using either the full batch of expressions (**A** and **C**) or the top ε fraction of the batch (**B** and **D**). Black plots (**A** and **B**) were trained using the risk-seeking policy gradient objective. Blue plots (**C** and **D**) were trained using the standard policy gradient objective. Colorbars indicate training step. Triangle markings denote the empirical mean of the distribution at the final training step. **E**. Training curves for mean reward of full batch (dotted), mean reward of top ε fraction of the batch (dashed), and best expression found so far (solid), averaged over all training runs.

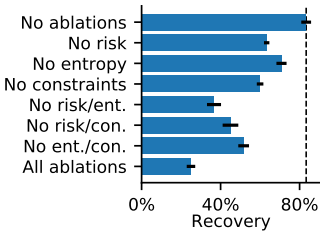

Figure 3: Recovery for various ablations of Algorithm 1 across all Nguyen benchmarks. Error bars represent standard error.

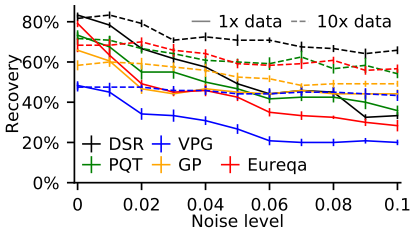

Figure 4: Recovery vs dataset noise and dataset size across all Nguyen benchmarks. Error bars represent standard error.

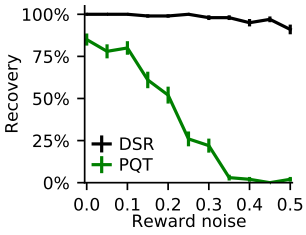

Figure 5: Recovery vs added reward noise on Nguyen-4. Error bars represent standard error.

smoothing the reward function. Thus, we repeated the noise experiments with 10-fold larger training datasets (Figure 4, dashed lines). For each dataset size, DSR consistently outperforms all baselines.

Performance under reward noise. The risk-seeking policy gradient is derived from a conditional expectation, and is thus well-justified for tasks with stochastic rewards. In contrast, PQT assumes deterministic rewards. Since symbolic regression is a deterministic task, we emulated a stochastic reward function by adding independent Gaussian noise directly to the reward function: $R'(\tau) = R(\tau) + \mathcal{N}(0, \sigma)$. In Figure 5, we compare DSR and PQT performance under increasing reward noise on benchmark Nguyen-4, a task that is an easier exploitation problem but difficult exploration problem. As reward noise increases, recovery for PQT heavily relies on exploration and luck, whereas DSR recovery remains stable even for high reward noise.

5 Conclusion and Future Work

We introduce a reinforcement learning approach to symbolic regression that outperforms state-of-the-art baselines in its ability to recover exact expressions on benchmark tasks. Our framework is easily extensible to other domains, which we save for future work; for example, searching the space of expressions to be used as control policies in reinforcement learning environments, or searching the space of organic molecular structures for high binding affinity to a reference compound. Our risk-seeking policy gradient formulation can also be applied to more traditional reinforcement learning domains; for example, optimizing for a high score (instead of average score) in Atari video games.

REFERENCES

Daniel A Abolafia, Mohammad Norouzi, Jonathan Shen, Rui Zhao, and Quoc V Le. Neural program synthesis with priority queue training. *arXiv preprint arXiv:1801.03526*, 2018.

Thomas Bäck, David B Fogel, and Zbigniew Michalewicz. *Evolutionary Computation 1: Basic Algorithms and Operators*. CRC press, 2018.

Irwan Bello, Barret Zoph, Vijay Vasudevan, and Quoc V Le. Neural optimizer search with reinforcement learning. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pp. 459–468. JMLR. org, 2017.

BGW Craenen, AE Eiben, and E Marchiori. How to handle constraints with evolutionary algorithms. *Practical Handbook Of Genetic Algorithms: Applications*, pp. 341–361, 2001.

Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. Robustfill: Neural program learning under noisy i/o. *arXiv preprint arXiv:1703.07469*, 2017.

Roger Fletcher. *Practical Methods of Optimization*. John Wiley & Sons, 2013.

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(Jul):2171–2175, 2012.

Giorgia Fortuna. Automatic formula discovery in the wolfram language. In *Wolfram Technology Conference 2015*. Wolfram Research, 2015. URL `https://library.wolfram.com/infocenter/Conferences/9329/`.

Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. *arXiv preprint arXiv:1801.01290*, 2018.

Quang Nhat Huynh, Hemant Kumar Singh, and Tapabrata Ray. Improving symbolic regression through a semantics-driven framework. In *2016 IEEE Symposium Series on Computational Intelligence (SSCI)*, pp. 1–8. IEEE, 2016.

Ying Jin, Weilin Fu, Jian Kang, Jiadong Guo, and Jian Guo. Bayesian symbolic regression. *arXiv preprint arXiv:1910.08892*, 2019.

John R Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*, volume 1. MIT press, 1992.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pp. 1097–1105, 2012.

Matt J Kusner, Brooks Paige, and José Miguel Hernández-Lobato. Grammar variational autoencoder. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pp. 1945–1954. JMLR. org, 2017.

Xin Li, Chi Zhou, Weimin Xiao, and Peter C Nelson. Prefix gene expression programming. In *Late breaking paper at Genetic and Evolutionary Computation Conference (GECCO2005), Washington, DC, USA*, pp. 25–29, 2005.

Chen Liang, Mohammad Norouzi, Jonathan Berant, Quoc Le, and Ni Lao. Memory augmented policy optimization for program synthesis with generalization. *arXiv preprint arXiv:1807.02322*, 2018.

Qiang Lu, Jun Ren, and Zhiguang Wang. Using genetic programming with prior formula knowledge to solve symbolic regression problem. *Computational intelligence and neuroscience*, 2016, 2016.

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. *PeerJ Computer Science*, 3:e103, January 2017. ISSN 2376-5992. doi: 10.7717/peerj-cs.103. URL `https://doi.org/10.7717/peerj-cs.103`.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529, 2015.

Aaron van den Oord, Sander Dieleman, Heiga Zen, Karen Simonyan, Oriol Vinyals, Alex Graves, Nal Kalchbrenner, Andrew Senior, and Koray Kavukcuoglu. Wavenet: A generative model for raw audio. *arXiv preprint arXiv:1609.03499*, 2016a.

Aaron van den Oord, Nal Kalchbrenner, and Koray Kavukcuoglu. Pixel recurrent neural networks. *arXiv preprint arXiv:1601.06759*, 2016b.

Aravind Rajeswaran, Sarvjeet Ghotra, Balaraman Ravindran, and Sergey Levine. Epopt: Learning robust neural network policies using model ensembles. *arXiv preprint arXiv:1610.01283*, 2016.

Prajit Ramachandran, Barret Zoph, and Quoc V Le. Searching for activation functions. *arXiv preprint arXiv:1710.05941*, 2017.

Sebastian Riedel, Matko Bosnjak, and Tim Rocktäschel. Programming with a differentiable forth interpreter. *CoRR, abs/1605.06640*, 2016.

Subham S Sahoo, Christoph H Lampert, and Georg Martius. Learning equations for extrapolation and control. *arXiv preprint arXiv:1806.07259*, 2018.

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

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

Aviv Tamar, Yonatan Glassner, and Shie Mannor. Policy gradients beyond expectations: Conditional value-at-risk. *arXiv preprint arXiv:1404.3862*, 2014.

Leonardo Trujillo, Luis Muñoz, Edgar Galván-López, and Sara Silva. neat genetic programming: Controlling bloat naturally. *Information Sciences*, 333:21–43, 2016.

Silviu-Marian Udrescu and Max Tegmark. Ai feynman: A physics-inspired method for symbolic regression. *Science Advances*, 6(16):eaay2631, 2020.

Nguyen Quang Uy, Nguyen Xuan Hoai, Michael ONeill, Robert I McKay, and Edgar Galván-López. Semantically-based crossover in genetic programming: application to real-valued symbolic regression. *Genetic Programming and Evolvable Machines*, 12(2):91–119, 2011.

David R White, James Mcdermott, Mauro Castelli, Luca Manzoni, Brian W Goldman, Gabriel Kronberger, Wojciech Jaśkowski, Una-May OReilly, and Sean Luke. Better gp benchmarks: community survey results and proposals. *Genetic Programming and Evolvable Machines*, 14(1): 3–29, 2013.

Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256, 1992.

Jiaxuan You, Rex Ying, Xiang Ren, William L Hamilton, and Jure Leskovec. Graphrnn: Generating realistic graphs with deep auto-regressive models. *arXiv preprint arXiv:1802.08773*, 2018.

Kaicheng Yu, Christian Sciuto, Martin Jaggi, Claudiu Musat, and Mathieu Salzmann. Evaluating the search phase of neural architecture search. *arXiv preprint arXiv:1902.08142*, 2019.

Barret Zoph and Quoc V Le. Neural architecture search with reinforcement learning. *International Conference on Learning Representations*, 2017.