DEEP SYMBOlic Regression: Recovering Mathematical Expressions From Data via Risk-Seeking Policy Gradients

Brenden K. Petersen
Lawrence Livermore National Laboratory
Livermore, CA, USA
bp@llnl.gov

Mikel Landajuela Larma
Lawrence Livermore National Laboratory
Livermore, CA, USA
landajuelalal1@llnl.gov

T. Nathan Mundhenk
Lawrence Livermore National Laboratory
Livermore, CA, USA
mundhenk1@llnl.gov

Claudio Santiago
Lawrence Livermore National Laboratory
Livermore, CA, USA
santiago10.gov

Sookyung Kim
Lawrence Livermore National Laboratory
Livermore, CA, USA
kim79@llnl.gov

Joanne T. Kim
Lawrence Livermore National Laboratory
Livermore, CA, USA
kim102@llnl.gov

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 \(f : \mathbb{R}^n \rightarrow \mathbb{R}\) 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 \cite{Schmidt2009}); 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 \cite{Yu2019}.

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 \cite{Lu2016}. Given this large, combinatorial search space, traditional approaches to symbolic regression typically utilize evolutionary algorithms, especially genetic programming (GP) \cite{Koza1992,Schmidt2009,Baeck2018}. 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 \cite{Krizhevsky2012} to optimal control \cite{Mnih2015}. 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 \cite{Udrescu2020} 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, \cite{Kusner2017} 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. \cite{Sahoo2018} 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 \ref{app_dsr} and \ref{app_related}.

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) \cite{Abolafia2018,Devlin2017,Riedel2016}. 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 $\tau$ by the pre-order traversal of its corresponding expression tree. We denote the $i$th token of the traversal as $\tau_i$ and the length of the traversal as $|\tau| = T$. Each token has a value within a given library $L$ of possible tokens, e.g. $\{+, -, \times, \div, \sin, x\}$.
We generate expressions one token at a time along the pre-order traversal (from $\tau_1$ to $\tau_T$). A categorical distribution with parameters $\psi$ 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 $\theta$ that emits a probability vector $\psi$ in an autoregressive manner. Specifically, the $i^{th}$ 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 $i^{th}$ token $\tau_i$, 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 $\tau_i$. The likelihood of the entire sampled expression is simply the product of the likelihoods of its tokens: $p(\tau|\theta) = \prod_{i=1}^{T} p(\tau_i|\tau_{1:(i-1)}; \theta) = \prod_{i=1}^{T} \psi^{(i)}_{\mathcal{L}(\tau_i)}$.

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 representation of the parent and sibling nodes of the token being sampled 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, \( \sigma_y \). That is, given a dataset \( (X, y) \) of size \( n \) and candidate expression \( f \), \( \text{NRMSE} = \frac{1}{\sigma_y} \sqrt{\frac{1}{n} \sum_{i=1}^{n} (y_i - f(X_i))^2} \). 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 \( \xi \) of the symbolic expression, which we optimize by maximizing the reward function: \( \xi^* = \arg \max_\xi R(\tau; \xi) \), 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: \( J_{\text{std}}(\theta) \doteq \mathbb{E}_{\tau \sim p(\tau|\theta)} [R(\tau)] \). The standard REINFORCE policy gradient (Williams, 1992) can be used to maximize this expectation via gradient ascent:

\[
\nabla_\theta J_{\text{std}}(\theta) = \nabla_\theta \mathbb{E}_{\tau \sim p(\tau|\theta)} [R(\tau)] = \mathbb{E}_{\tau \sim p(\tau|\theta)} [R(\tau) \nabla_\theta \log p(\tau|\theta)]
\]

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 \( \varepsilon \):

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

(1)

This objective aims to increase the reward of the top \( \varepsilon \) fraction of samples from the distribution, without regard for samples below that threshold. This objective bears close resemblance with \( \varepsilon \)-conditional value at risk (CVaR), for which the “\( \leq \)” symbol is used instead of “\( \geq \)” and the \( \varepsilon \)-quantile of rewards is used instead of the \((1 - \varepsilon)\)-quantile. Optimizing CVaR is a form of risk-averse learning.
We compared DSR against five strong symbolic regression baselines: (1) \( \tilde{J} \) where we show the analogous policy gradient of Algorithm 1

Evaluating DSR.

our framework trained using priority queue training (Abolafia et al., 2018) in place of the risk-seeking used to determine whether the best found candidate expression was correctly recovered.

evaluate the best found candidate expression at the end of training, and the ground truth expression is

expressions. Pseudocode for DSR is shown in Algorithm 1. Source code is made available at

the approach used to optimize CVaR (Tamar et al., 2014; Rajeswaran et al., 2016) for risk-averse

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

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 \( J_{\text{risk}}(\theta; \varepsilon) \) denote the conditional expectation of rewards above the \((1 - \varepsilon)\)-quantile, as in Equation (7). Then the gradient of \( J_{\text{risk}}(\theta; \varepsilon) \) is given by:

\[
\nabla_{\theta} J_{\text{risk}}(\theta; \varepsilon) = E_{\tau \sim p(\tau | \theta)} [(R(\tau) - R_{c}(\theta)) \cdot \nabla_{\theta} \log p(\tau | \theta) \mid R(\tau) \geq R_{c}(\theta)]
\]

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:

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

where \( \tilde{R}_{c}(\theta) \) is the empirical \((1 - \varepsilon)\)-quantile of the batch of rewards, and \( 1_{x} \) 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, \( R_{c}(\theta) \), whereas the baseline for standard policy gradients is non-specific, chosen by the user; and (2) effectively, only the top \( \varepsilon \) 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 \( \varepsilon \) 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 https://github.com/brendenpetersen/deep-symbolic-regression

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 (Abolaña et al., 2018) in place of the risk-seeking

```python
Algorithm 1 Deep symbolic regression with risk-seeking policy gradient

input learning rate \( \alpha \); entropy coefficient \( \lambda_H \); risk factor \( \varepsilon \); batch size \( N \); reward function \( R \)

output Best fitting expression \( \tau^* \)

1: Initialize RNN with parameters \( \theta \), defining distribution over expressions \( p(\cdot | \theta) \)
2: \( \tau \leftarrow \{ \tau^{(i)} \sim p(\cdot | \theta) \}_{i=1}^{N} \) \quad \text{Sample } \( N \) expressions (Alg. 2 in Appendix A)
3: \( \tau \leftarrow \{ \text{OptimizeConstants}(\tau^{(i)}, R) \}_{i=1}^{N} \) \quad \text{Optimize constants w.r.t. reward function}
4: \( \tau \leftarrow \{ R(\tau^{(i)}) \}_{i=1}^{N} \) \quad \text{Compute rewards}
5: \( R_{c} \leftarrow (1 - \varepsilon)\)-quantile of \( \tau \) \quad \text{Compute reward threshold}
6: \( \tau \leftarrow \{ \tau^{(i)} : R(\tau^{(i)}) \geq R_{c} \} \) \quad \text{Select subset of expressions above threshold}
7: \( \tau \leftarrow \{ \text{ReduceMean}(\{ R - R_{c} \} \nabla_{\theta} \log p(T | \theta)) \} \quad \text{Compute risk-seeking policy gradient}
8: \( \tau \leftarrow \{ \text{ReduceMean}(\nabla_{\theta} \frac{\lambda_H}{N} H(T | \theta)) \} \quad \text{Compute entropy gradient}
9: \( \theta \leftarrow \theta + \alpha (\hat{g}_1 + \hat{g}_2) \quad \text{Apply gradients}
10: \text{if } \max \tau > R(\tau^*) \text{ then } \tau^* \leftarrow \tau(\text{arg max } \tau) \quad \text{Update best expression}
11: \text{return } \tau^*
```

Published as a conference paper at ICLR 2021
Wolfram results are only presented for one-dimensional benchmarks because the method is not applicable to higher dimensions. All experiments were replicated with 100 different random seeds for each benchmark expression.

For comparison, the best-case performance of the standard policy gradient plateaus early in training (Figure 2). 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 $\varepsilon$ 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 2)

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.)

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 F, we report recovery on several additional variants of Nguyen-8. (Analogous curves are provided in Appendix F.)

Each benchmark suite. A bold value represents statistical significance ($p < 10^{-3}$) across all benchmarks.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Expression</th>
<th>DSR</th>
<th>PQT</th>
<th>VPG</th>
<th>GP</th>
<th>Eureqa</th>
<th>Wolfram</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nguyen-1</td>
<td>$x^2 + x^3 + x$</td>
<td>100%</td>
<td>100%</td>
<td>96%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>Nguyen-2</td>
<td>$x^4 + x^3 + x^2 + x$</td>
<td>100%</td>
<td>99%</td>
<td>47%</td>
<td>97%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>Nguyen-3</td>
<td>$x^5 + x^4 + x^3 + x^2 + x$</td>
<td>100%</td>
<td>92%</td>
<td>4%</td>
<td>100%</td>
<td>95%</td>
<td>100%</td>
</tr>
<tr>
<td>Nguyen-4</td>
<td>$x^6 + x^5 + x^4 + x^3 + x^2 + x$</td>
<td>100%</td>
<td>93%</td>
<td>1%</td>
<td>100%</td>
<td>70%</td>
<td>100%</td>
</tr>
<tr>
<td>Nguyen-5</td>
<td>$\sin(x^4) \cos(x) - 1$</td>
<td>72%</td>
<td>73%</td>
<td>5%</td>
<td>45%</td>
<td>73%</td>
<td>2%</td>
</tr>
<tr>
<td>Nguyen-6</td>
<td>$\sin(x) + \sin(x + x^2)$</td>
<td>100%</td>
<td>98%</td>
<td>100%</td>
<td>91%</td>
<td>100%</td>
<td>1%</td>
</tr>
<tr>
<td>Nguyen-7</td>
<td>$\log(x + 1) + \log(x^2 + 1)$</td>
<td>35%</td>
<td>41%</td>
<td>3%</td>
<td>0%</td>
<td>85%</td>
<td>0%</td>
</tr>
<tr>
<td>Nguyen-8</td>
<td>$\sqrt{x}$</td>
<td>96%</td>
<td>21%</td>
<td>5%</td>
<td>5%</td>
<td>0%</td>
<td>71%</td>
</tr>
<tr>
<td>Nguyen-9</td>
<td>$\sin(x) + \sin(y^2)$</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>–</td>
</tr>
<tr>
<td>Nguyen-10</td>
<td>$2\sin(x) \cos(y)$</td>
<td>100%</td>
<td>91%</td>
<td>99%</td>
<td>76%</td>
<td>64%</td>
<td>–</td>
</tr>
<tr>
<td>Nguyen-11</td>
<td>$x^y$</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>7%</td>
<td>100%</td>
<td>–</td>
</tr>
<tr>
<td>Nguyen-12</td>
<td>$x^4 - x^3 + \frac{1}{2} y^2 - y$</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>–</td>
</tr>
</tbody>
</table>

Average 83.6% 75.2% 46.7% 60.1% 73.9% –

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).
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 $\varepsilon$ 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 $\varepsilon$ 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.

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 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: $R'(\tau) = R(\tau) + N(0, \sigma)$. 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.

ACKNOWLEDGMENTS

We thank Ruben Glatt, Thomas Desautels, Priyadip Ray, David Widemann, and the 2019 UC Merced Data Science Challenge participants for their useful comments and insights. This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344. Lawrence Livermore National Security, LLC. LLNL-CONF-790457.

REFERENCES


