Symbolic Regression via Deep Reinforcement Learning Enhanced Genetic Programming Seeding

Anonymous Author(s)
Affiliation
Address
email

Abstract

Symbolic regression is the process of identifying mathematical expressions that fit observed output from a black-box process. It is a discrete optimization problem and generally believed to be NP-Hard. Some example use cases would be to automatically discover the underlying mathematical formulas that describe a physical system given a set of observations or for finding more easily explainable optimal controllers. Prior approaches to solving the issue include *genetic programming* and *deep reinforcement learning*. In this work, we introduce a hybrid approach that uses a policy gradient-based deep reinforcement learner (DRL) to enhance the seeding of a random-restart genetic programming-based symbolic regressor (GPSR) by gradually learning better starting populations samples. On a number of common benchmark tasks to recover underlying equations from a dataset, our method recovers 65% more equations than the most recently published top-performing model using the same experimental setup. We suggest that running many GPSR steps independently of the DRL is better for symbolic regression than a method where the two are more interdependent. Finally, we introduce a new benchmark set of 22 expressions with increased difficulty over existing benchmarks. Source code is provided at *www.anonymous.submission*.

1 Introduction

Symbolic regression involves fitting a dataset with equations which are potentially easier to interpret than, for example, neural networks. It is different than fitting data with a polynomial or a deep neural network because we want to illuminate the true underlying process that generated the data. The process of symbolic regression is analogous to how a physicist may derive a set of fundamental equations to describe a natural process. For example, Tycho Brahe meticulously mapped the motion of planets through the sky, but it was Johannes Kepler who later created the equations for the laws that described their motion. Given a dataset (X, y), where each point with parameters $X_i \in \mathbb{R}^n$ and observations $y_i \in \mathbb{R}$, we aim 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 closed-form mathematical expression.

The space of mathematical expressions is structurally discrete but continuous in parameter space. The space grows exponentially with the length of the expression, rendering symbolic regression a challenging machine learning problem. It is believed to be NP-hard [Lu et al., 2016]; however, no formal proof exists. Given the large, combinatorial search space, traditional approaches to symbolic regression commonly utilize evolutionary algorithms, especially *genetic programming* (GP) [Koza, 1992, Schmidt and Lipson, 2009, Fortin et al., 2012, Bäck et al., 2018]. GP-based symbolic regression operates by maintaining a population of mathematical expression "individuals" that are "evolved" using evolutionary operations like selection, crossover, and mutation. A fitness function acts to improve the population over many generations.

Submitted to 35th Conference on Neural Information Processing Systems (NeurIPS 2021). Do not distribute.

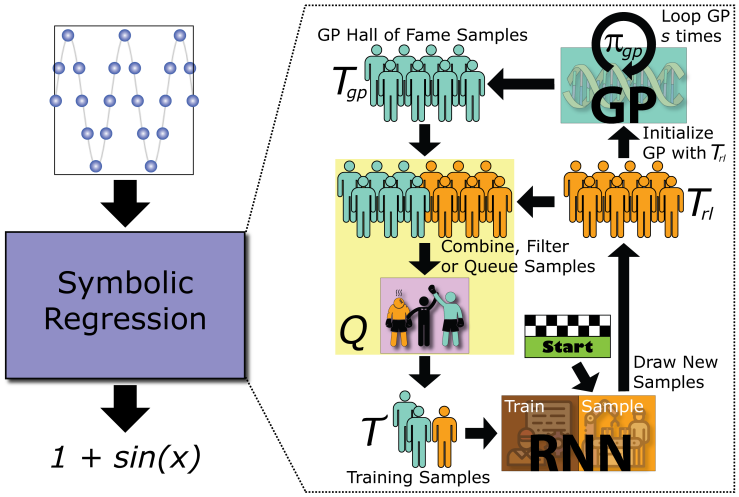

Figure 1: Our approach works by using a policy-gradient trained RNN to generate samples of expressions which might solve our symbolic regression problem. These individuals in evolutionary parlance, are used to seed a GP solver. It will run several times and return a set of the most fit individuals. A hall-of-fame contains the most fit expressions over all GP generations, in the current loop. These are merged with the RNN created individuals and used to update the RNN. The GP is stateless and runs in a random-restart like manner that is seeded by the RNN. The process repeats until either the correct expression is found or we reach our hard limit of evaluations. More details can be seen later on in algorithm 1.

Neural networks can also be leveraged for symbolic regression [Kusner et al., 2017, Sahoo et al., 2018, Udrescu and Tegmark, 2020]. Recently it has been proposed to solve symbolic regression using *deep reinforcement learning* (DRL); specifically, policy gradients [Petersen et al., 2021, Kim et al., 2021]. This approach works by using a *recurrent neural network* (RNN) to stochastically emit batches of equations as a sequence of mathematical operators or "tokens." Expressions are evaluated for goodness of fit and a *risk-seeking* strategy is used to filter out lesser performers and return a better set of equations to the RNN for training. Additionally, constraints can be employed to prune the search space, preventing nonsensical or extraneous statements in generated expressions, for instance, inversions (*e.g.* $\log(e^x)$) and nested trigonometric functions (*e.g.* $\sin(1+\sin(x))$). The performance attained using DRL outperformed GP-based approaches, including commonly used commercial software while also leading to concise, tractable, and more readily interpretable solutions. However, the problem is not yet solved and pure DRL still struggles with harder expressions.

Genetic programming and policy gradient search are mechanistically dissimilar, yet both have proven to be effective solutions to symbolic regression. Might it be possible to combine the two approaches to leverage each of their strengths? The population of individual solutions in GP is structurally the same as a batch of samples emitted by the RNN in policy-gradient search. In principle, they can formally be interfaced by allowing GP individuals to flow to the RNN and vice versa. This way of coupling creates an out-of-distribution problem on the RNN side that alters the optimization of the classical RL objective, i.e., maximization of expected return. Note that standard off-policy methods, like importance sampling [Glynn and Iglehart, 1989], do not apply here since the GP distribution is not tractable. However, the mechanistic interpretation of the policy-gradient as maximizing the log likelihood of individuals scaled by their reward is still valid. Following this latter interpretation, in this paper we investigate this simple two-way coupling without any off-policy correction.

In our approach, we view the policy-gradient as much more capable of nuance, perhaps even when given off-policy training samples. Its stochastic nature can help to avoid many local minima, but is most likely still prone to them. GP on the other hand was designed to jump over local minima. While it will still seek a minimum in a manner comparable to policy-gradient [Such et al., 2018], it is unclear how well it can move in subtle ways. What we want is the ability to still jump over local minima like a GP, but be able to seek and settle like a policy-gradient.

Our goal is to avoid changing the typical behavior of the GP as much as possible for the RL seeding to be effective. The technique we propose to accomplish this is to iteratively improve the individual

population seeding of the GP from the policy-gradient RNN which learns to create better seeds. Essentially, this prevents us from making internal changes to the GP black box and how it works over several generations. Other than the change of the input, which may not have a policy for GP, it will work as usual.

2 Related Work

Deep learning for symbolic regression. Several recent approaches leverage deep learning for symbolic regression. AI Feynman [Udrescu and Tegmark, 2020] proposes 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. It can be thought of as a meta-solver that integrates other solvers together rather than being a solver itself. 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 equations 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, back-propagation 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 $\sin(x/y)$.

Genetic programming/policy gradient hybrids. The overall idea of combining GP with gradient-based methods in general predates the deep learning era [Topchy and Punch, 2001, Zhang and Smart, 2004, Montastruc et al., 2004, Wierstra et al., 2008] and dissatisfaction with pure DRL is not entirely new [Kurenkov, 2018]. Recently, several approaches have combined GP and *reinforcement learning* (RL). In these works, RL is used to alter or augment the GP process, e.g. by adjusting the probabilities of performing different genetic operations. Alternatively, GP is used to augment the creation or operation of a neural network [Such et al., 2018, Chang et al., 2018, Gangwani and Peng, 2018, Chen et al., 2019, Stanley et al., 2019, Real et al., 2019, Miikkulainen1 et al., 2019, Sehgal et al., 2019, Tian et al., 2020, Chen et al., 2020a,b]. The target application tends to be *network architecture search* (NAS). While NAS involves discrete optimization like symbolic regression, a critical difference is the time scale of individual evaluations and the size of resulting solution networks. In NAS, it is sufficient to find a better solution since the evaluation takes much longer and the obtained network can involve billions of parameters. Given our quicker evaluations and dramatically smaller solution size, we can use algorithms more likely to find the one exact solution.

Sample population interchanging methods. We identify a broad class of existing methods that can be characterized by using samples from a *sequence generator* (SG) and interchanging those samples with a GP population. The SG can be any discrete distribution or generative process that creates a sequence of tokens, such as a reinforcement learner, recurrent neural network, or deep transformer. A set of SG-generated samples is then treated as interchangeable with a population generated by GP. Thus, SG samples can be inserted into the population of GP, and GP samples can be used to update the SG. In some cases the SG may not need learnable parameters.

Within this class of methods, Pourchot and Sigaud [2019] and Khadka and Tumer [2018] solve physical continuous control problems like the inverted pendulum or moon rover with a fungible neural network controller. Their approaches are similar to Neuroevolution [Stanley and Miikkulainen, 2002, Floreano et al., 2008, Lüders et al., 2017, Risi and Togelius, 2017], as both techniques are based on using deep deterministic policy gradients (DDPG) [Lillicrap et al., 2016] and utilize a synchronous shared pool of actors between GP and RL components. Khadka and Tumer [2018] introduced the *ERL* method that pools RL and GP actor samples into a cyclic replay buffer. This single buffer is drawn to either seed a GP step or help update the actor-critic model. Both GP and RL work synchronously and the population is interchangeable. Pourchot and Sigaud [2019] introduce *CEM-RL* which solves the same family of problems in a similar way. The most notable difference is that the shared population is not fed directly into a GP, but rather it is used to update a distribution $\mathcal{N}(\pi_\mu, \Sigma)$ from which a population is drawn.

In particular, within this class of methods Ahn et al. [2020] develop genetic expert-guided learning (GEGL), which exhibits several similarities to our own. The authors address the problem of *de novo* molecular design [Schneider and Fechner, 2005, Schneider, 2013, Popova et al., 2018] to discover novel molecules for fields such as pharmacy [Hughes et al., 2011, Minnich et al., 2020, Jacobs et al.,

2021] and materials science [Yan et al., 2018]. In GEGL, samples are generated using a stochastic RNN-based SG called an *apprentice*. All samples are added to a max-reward priority queue (MRPQ). The GP component comprises two evolutionary operators (mutation and crossover) performed on each item returned by the MRPQ. Interestingly, mutation is only performed if crossover fails to create a viable individual after multiple attempts. Since only one generation of evolutionary operators is applied for each batch of SQ samples, there is no notion of a selection operator. In contrast, we have found that the selection operator and multiple generations of evolution is crucial to maximize the benefits of GP. All individuals from the GP population are then added to a second MRPQ. The union of the GP-based MRPQ and SG-based MRPQ are then passed to the apprentice for training.

We find in our approach that it is unclear if any priority queues are necessary. As we will discuss later, whether or not we use PQ, does not show a significant difference. As with ERL and CEM-RL, GEGL's evolutionary and reinforcement method steps are 1-to-1, meaning that each RL step is followed by exactly one GP generation. Additionally, GP pulls from a continuous memory of sample population which it shares with the RL component. This strong coupling makes the evolutionary component much more interdependent on the neural network and changes GP core behavior.

3 Methods

We are interested in generating expressions and testing them for goodness of fit. We do this by stringing together tokens that can represent a variable, constant or function (e.g. $+, -, \sin, \exp$). In policy gradient search, an RNN generates expressions by emitting tokens sampled one-at-a-time from a distribution and attaching them to a string $t^{(i)}$. These strings are part of a sample batch T_{rl} of size N. Each sample equation is executed and assigned a reward function $R(\tau)$ based on how well it fits the data. It is typical to use inverse normalized root-mean-squared error (NRMSE) to compute reward over a dataset (X, y) of size n and candidate expression f as NRMSE $= \frac{1}{\sigma_y}\sqrt{\frac{1}{n}\sum_{i=1}^{n}\left(y_i - f\left(X_i\right)\right)^2}$ with σ_y being the standard deviation of y. This is done simply by feeding each data point X_i into the expression f and subtracting it from y_i. The sample rewards are then used to update the RNN parameters θ using the policy gradient formula:

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

Ideally every time we update the RNN, the distribution of samples will become more like the expressions with higher reward. The samples τ used in the update can be redacted prior to their usage. *Priority queue training* (PQT) [Abolafia et al., 2018] places a PQ between the reward evaluation and update. This works by putting the samples and rewards into a fixed size queue sorted by reward. A set number of the best samples are used to update the RNN. The memory of the queue is maintained and the RNN can sometimes be updated by samples several batch steps in the past. The reason for holding over higher rewarding samples is intuitive. However, we no longer have a policy-gradient as it is usually defined. We can also use a *risk seeking policy gradient* (RSPG) [Petersen et al., 2021] that removes the bottom ϵ percent of expressions from update. For notational purpose, when we do no redacting of samples, this is called *vanilla policy gradient* (VPG).

Genetic Programming starts with a population of individuals T_{gp}. Usually these are randomly generated, however, in our case they are generated by the RNN. For each individual, there is a set probability included in η that it will be mutated or mated (crossover) at any generation. Mutation is achieved by replacing nodes or branches of the expression tree with something random, yielding a new expression somewhat like the old one but with some aleatory changes. Through this process, we generate mutations that are serendipitous and effectively create expressions which better fit our data in some cases even though many are worse. Expressions are mated by taking some parts from one expression and parts from another one and creating a new expression which is a valid synthesis of the two. We aim for the offspring to have qualities of its parents which are good and as a result make it better than its parents. Much like with mutation, the likelihood is at least as good that the resultant is worse. To move the population in a desirable direction towards the solution, we remove less *fit* individuals and try to keep the better ones. Fitness in our case is measured by *normalized mean-squared error* NMSE $= \frac{1}{n}\sum_{i=1}^{n}\left(y_i - f\left(X_i\right)\right)^2$.

After many generations, the population will fit the data better overall, and be more likely to give rise to even more fit individuals. To maintain diversity, we weed out individuals by holding a large number of fixed size small tournaments of size Z in which only the winner is kept. Since we have

Algorithm 1 Symbolic Regression via Deep Reinforcement Enhanced Genetic Seeding (DREGS)

input *All* $\rightarrow$ batch size N; reward function R; GP steps S; constraints Ω
input *All* $\rightarrow$ Sample selection queuing function Q. Can be used with PQT, RSPG, or neither.
input *All* $\rightarrow$ Sample selection size M. Note that M can be equal to N.
input *GP* $\rightarrow$ crossover probability P_c; mutation probability P_m; tournament size Z
input *RL* $\rightarrow$ learning rate α; entropy coefficient $\lambda_{\mathcal{H}}$
output Best fitting expression $\tau^\star$

define θ parameters for reinforcement learning
define η parameters for genetic programming
define a set of expressions $\mathcal{T}$ with data structure to be determined
define R NN distribution over expressions $p(\cdot|\theta)$
  Initialize RNN with parameters $\theta, \alpha, \lambda_{\mathcal{H}}$
  $\eta \leftarrow \{P_c, P_m, Z\}$ ▷ Set GP Parameters
  $Q \leftarrow ClearSamples\,(Q)$ ▷ Init function Q with no samples
  loop
    $T_{rl} \leftarrow \{t^{(i)} \sim p(\cdot|\theta, \Omega)\}_{i=1}^{N}$ ▷ Sample N RNN expressions with constraints Ω
    $T_{gp} \leftarrow T_{rl}$ ▷ Copy expressions for GP
    $\pi_{gp} \leftarrow InitGpPolicy\,(T_{gp}, \eta, R)$ ▷ Reset GP policy seeded from RL expressions
    for $s = 1, \ldots, S$ **do**
      $T_{gp} \leftarrow \pi_{gp}\,(T_{gp}, \Omega)$ ▷ Run a GP step and update expression population
    $\mathcal{T} \leftarrow Q\,(T_{gp} \cup T_{rl}, R)$ ▷ sample selection and return M expressions
    $\mathcal{R} \leftarrow \{R(\tau^{(i)})\}_{i=1}^{M}$ ▷ Gather rewards from expressions
    $\theta \leftarrow UpdateWeights\,(\theta, \mathcal{T}, \mathcal{R})$ ▷ Update the RNN parameters via policy gradient
    if not keeping samples **then** $Q \leftarrow ClearSamples\,(Q)$ ▷ *e.g.* PQT maintains samples
    if $\max \mathcal{R} > R(\tau^\star)$ **then** $\tau^\star \leftarrow \tau^{(\arg\max \mathcal{R})}$ ▷ Update best expression
return $\tau^\star$

some probability of having tournaments with only unfit individuals, there is some chance that they can endure for more generations and keep the gene pool diverse. GP does not have an explicit mathematical imperative to find the highest reward like policy-gradient, but it does tend to move faithfully in that direction [Such et al., 2018].

For both GP and policy-gradients, constraints are applied to prevent absurd expressions or to limit resource utilization. A common constraint is to limit the maximum size of an expression. Specialized constraints can also be used related to the type of problem we are solving. Since we are trying to create usable equations, we can put a restriction in place that an expression is properly formed and does not have left over dangling terminals. We can apply the constraints on GP individuals by not allowing violators to be kept in the population. For policy-gradients, we can adjust distributional likelihoods to zero to eliminate it producing an invalid expression at any given point as we draw tokens to create a sample.

Our policy gradient mechanism is the same as Petersen et al. [2021] with the notable exception that we interface it with GP. As a base we start with the symbolic regression GP solution from Fortin et al. [2012] with a few key changes. These are:

1. Typically only *uniform* mutation is used. Instead we use one of uniform, node replacement, insertion or shrink mutation with equal probability when an individual is mutated.
2. We use the extra constraints from Petersen et al. [2021] that remove nested trigonometric functions (*e.g.* $\sin(\sin(\sin(x)))$) and inverting functions (*e.g.* $\log(e^x)$). Any individual that is created that violates the constraints is never kept.
3. The sample population is never initiated randomly. It is always seeded from the policy-gradient RNN.
4. We use a tournament size of 5 rather than 3. This decreases the likelihood of unfit individuals making it to the next generation.

Our method is formulated in algorithm 1 and is sketched out in figure 1. We will refer to it as *Deep Reinforcement Enhanced Genetic Seeding* (DREGS). The notion is that the GP should be allowed to run as it would normally, but stably be placed closer to the solution over time at each start. That is, we want the policy-gradient to act as a stabilizing and guiding factor without changing the way the

GP works. The stability component is possibly the most critical since it is unclear how stable GPs are, but we are fairly certain that they move in the direction of optimal fitness (reward). Policy-gradients can be unstable if parameters are chosen wrong, but we should expect it to eventually settle very close to a minima. As such, the policy-gradient will act to seed the GP which will run for many steps before ending. As the RNN becomes biased towards a solution state, the GP can still perform a large number of permutations on those seeds and create a high diversity of solutions. The GP solutions are then fed back to the policy-gradient RNN at update for training. This means that if the GP has an instance of revelation, the update can begin to move the RNN in a new direction since the GP individuals are *off-policy* training solutions. As such, we would expect this to make the RNN more exploratory and less likely to get stuck on a local minima. This also means that the RNN may serve to only optimize the results of the GP.

The RNN goes first and in its untrained state, it will create a mostly random set of starter individuals T_{rl}. The RNN is still bound by constraints Ω, so the seeds are somewhat less randomized than they otherwise would be. The GP is reset from these new seeds and is essentially stateless. That is, it never retains any individuals between RNN batch selections. It is for all intents and purposes, running in a random-restart fashion. The only change over each GP generation is the updated population seeds provided by the RNN.

Given that each generation in a GP will make a set number of changes, the number of generations it should run each time is proportional to the expected length of the target expression. That is, given a seed individual A and the unknown solution expression B, the expected number of operations, and thus generations to transform A into B should be proportional to the hamming distance. This guides the parameter selection of *25 generations* s for each time the GP runs based on the estimation on how many steps it would generally take to transform an A into some more desirable B. We further illuminate this parameter choice in the discussion section (see also figure 2).

The best unique *N* hall-of-fame (best over all generations s in the loop) expressions T_{gp} are returned after s steps. These are merged back with the seeding expressions T_{rl} into container Q. We have explored three different container options for how to merge them together:

1. *PQT*: Combine T_{gp} and T_{rl} into a PQ container, where the top M expressions are returned to update the RNN.
2. *RSPG*: Use risk seeking policy gradient to only return the top ϵ percent of expressions to update the RNN.
3. *VPG*: Use a vanilla policy gradient that makes no extra selection. All expressions are returned.

We chose these types since they have been used in similar situations (*e.g* [Abolafia et al., 2018, Petersen et al., 2021]). Unless otherwise stated, *PQT* is the method we use for benchmarking.

Once evaluations are in Q, we can then pass them back to the policy-gradient RNN to update weights θ. After that, the loop repeats and we draw N new samples from the RNN and give them to the GP. If any expression has zero error, we can quit immediately and return it as the final solution. Alternatively, we can give up at a certain point after a set number of evaluations or time. For the sake of proper benchmarking with other solutions, we quit when we have reached 2 million evaluations. That is, once we have created and tested 2 million equations, we return the best expression (by error) seen of them all. Note that repeat expressions still count towards the total since we also want this limit to have a proportional nature to the amount of time we expect it to take. More details on our hyperparameter selection can be found in the appendix table 10.

4 Results

We used two popular benchmarks to compare our technique to other methods: Nguyen [Uy et al., 2014] and the R rationals [Krawiec and Pawlak, 2013]. Additionally, we created a new benchmark we are releasing with this paper. We had created our set we call *anonymous-submission-name* prior to this effort, but decided to use and release it as it seems apparent that the results on Nguyen are starting to have too little variance and R is too small. Sometimes we will note that we are using Nguyen-12*, R1*, R2* or R3*. In these four cases, we found that the solution was much more tractable if we expanded the range over which we draw samples. Without this modification, R1, R2 and Nguyen-12 are still intractable. Allowing these equations to be solved give us more variance in scoring and makes them more useful for benchmarking, in some cases. All equations are shown in the appendix

Table 1: DREGS is compared with several state-of-the-art symbolic regression algorithms. Shown is how often the method recovers the ground-truth expression for each of the 12 Nguyen benchmarks. This is done by running each technique 100 times with different random-number seeds each time. PQT and VPG are run as DRL without GP per [Petersen et al., 2021].

		% Recovery					
Benchmark	Expression	DREGS	DSR	PQT	VPG	GP	Eureqa
Nguyen-1	$x^3 + x^2 + x$	100	100	100	96	100	100
Nguyen-2	$x^4 + x^3 + x^2 + x$	100	100	99	47	97	100
Nguyen-3	$x^5 + x^4 + x^3 + x^2 + x$	100	100	86	4	100	95
Nguyen-4	$x^6 + x^5 + x^4 + x^3 + x^2 + x$	100	100	93	1	100	70
Nguyen-5	$\sin(x^2)\cos(x) - 1$	100	72	73	5	45	73
Nguyen-6	$\sin(x) + \sin(x + x^2)$	100	100	98	100	91	100
Nguyen-7	$\log(x + 1) + \log(x^2 + 1)$	97	35	41	3	0	85
Nguyen-8	$\sqrt{x}$	100	96	21	5	5	0
Nguyen-9	$\sin(x) + \sin(y^2)$	100	100	100	100	100	100
Nguyen-10	$2\sin(x)\cos(y)$	100	100	91	99	76	64
Nguyen-11	x^y	100	100	100	100	7	100
Nguyen-12	$x^4 - x^3 + \frac{1}{2}y^2 - y$	0	0	0	0	0	0
	Average	91.4	83.6	75.2	46.7	60.1	73.9

Table 2: We compare DREGS with DSR, GEGL and random-restart GP on the Nguyen, R, and Anonymous benchmark sets. The methodology is the same as seen in table 1. However for these results we only run each expression 25 times rather than 100. This is done primarily to save time. For each benchmark set, we can see the mean recovery rate as well as the recovery rate for all benchmarks combined. The 95% confidence interval is obtained from the standard error between mean recovery on 32 unique benchmarking runs. These have different conditions and parameters that were used to generate ablation results and other comparisons. Detailed results for each benchmark can be see in the appendix as table 5. All expressions can be seen in the appendix tables 8 and 9.

	Mean % Recovery			
	All	Nguyen	R	Anonymous
DREGS	74.92	92.33	33.33	71.09
GEGL [Ahn et al., 2020]	64.11	86.00	33.33	56.36
Random Restart (GP Only)	63.57	88.67	2.67	58.18
DSR (RL Only) [Petersen et al., 2021]	45.19	83.58	0.00	30.41
95% Confidence Interval $(n = 32)$	$\pm$ 1.543	$\pm$ 1.761	$\pm$ 2.809	$\pm$ 1.321

(see tables 8 and 9). We used the benchmark Nguyen-7 to work out hyperparameter settings for our GP component. We used Nguyen-7 and R3* to work out hyperparameters for our policy gradient component. None of the *anonymous-submission-name* equations were used for parameter tuning.

Experiments were conducted on 36 core, 2.1 GHz, Intel Xeon E5-2695 workstations. Our implementation is CPU only. Each benchmark that is run, is only executed on a single machine. Full benchmarking computes the 37 test cases across one of 10 identical machines. We follow the testing procedure of [Petersen et al., 2021] unless otherwise noted. For all benchmark tests, each one is run several times given a unique random number seed. Our primary benchmark is how often a solution is found for any given equation before we reach the maximum 2 million evaluations. The ground truth expression is used to determine whether the best found candidate was correctly recovered. Table 1 shows how often our method recovers the current equation on the Nguyen benchmark suite compared with DSR, PQT, VPG [Petersen et al., 2021], GP [Fortin et al., 2012] and Eureqa [Schmidt and Lipson, 2009]. We note that DSR is the prior top performer on this set. For these experiments, each benchmark expression was tried 100 times using a different random number seed. As we can see, the recovery rate for DREGS is higher than it is for the current leader DSR.

For our next experiment we also compare with a method called GEGL [Ahn et al., 2020]. This was designed to find new molecules that fit some set of desired parameters. It is similar enough to our own method that it warranted comparison. The implementation that is available online is strictly

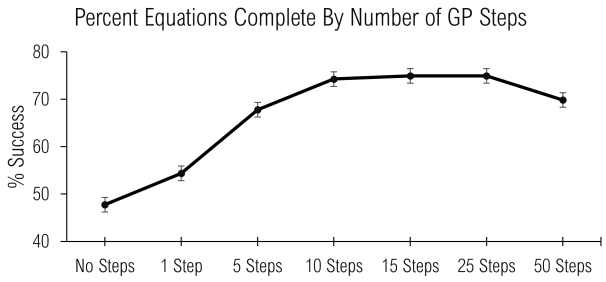

Figure 2: The percent recovery rate is shown over Nguyen, R and Anonymous benchmarks for 25 runs per benchmark as was done in 2. The number of steps is how many GP generation steps were run before returning samples and resetting. Error bars are at the 95% confidence interval previously used with $(n = 32)$. The *no steps* condition is the policy-gradient RNN (DSR) running all on its own without GP.

designed for solving the molecular problem of their interest. So, we adapted it to solve for symbolic regression and tuned the hyperparameters in the same way described prior. We also compare with a method which only used random-restart GP without the policy-gradient. We option to use R1*,R2* and R3* as well as use Nguyen-12* rather than Nguyen-12. For this experiment, we do 25 runs rather than 100 as in table 1. Finally we test against our own *anonymous* dataset. Table 2 shows that overall, DREGS outperforms the other three methods. DREGS also performs best on Nguyen and *anonymous* datasets. GEGL ties DREGS on the R benchmark. It is interesting to note how well random-restart GP does on its own. We should mention that compared with GP from [Fortin et al., 2012], this one also has the prior described GP features that DREGS has. That is, it is just DREGS without the policy-gradient part. Finally, DREGS outperforms the current leading DSR benchmark with an approximate 65% improvement in performance.

5 Discussion

We stated that one of our beliefs is that the number of GP generation steps that should run between each RL step is proportional to expression length. That is, the larger an expression is, the more evolutionary operations should be required to transform any arbitrary seed individual into the target expression. Our initial estimate was that 25 GP steps would be needed per each RL step. Figure 2 shows a post-hoc analysis of how performance varies depending on how many GP steps we do between each RL step. The optimal number of steps is between 10 and 25. We note that the hard limit set by our implementation is 30 tokens. However, each GP step can make many changes at a time, so we expect the number of needed GP steps to be less than the maximum possible hamming distance. As such, we can see the similarity between the optimal number of GP steps and the length of expressions.

Ablations. We ran a number of post-hoc ablations to try and determine the contribution of our various design choices to our methods success rate. Table 3 shows the recovery rate as a factor of different components. The first thing to note is that different choices of Q container do not make a large difference. While PQT outperforms VPG and RSPG, the difference is well within the 95% confidence interval $(n = 32)$. As such, it is unclear if any one is really superior to the other. Additionally, removing the common entropy term $\lambda_{\mathcal{H}} \nabla_\theta \mathcal{H}(\mathcal{T}|\theta)$ by setting $\lambda_{\mathcal{H}}$ to zero reduces performance by an amount with confidence intervals. We should not expect it to have a large effect since sampling diversity should be covered by the GP component.

Using different types of mutation in the GP component with equal probability helps by a small but probably significant amount. We had hypothesised that increasing the diversity of GP operations would equate to an increased variety of generated expressions. Using the RL constraints of [Petersen et al., 2021] during GP steps also seems to help. The two combined account for a 10% gain in performance. Next we wanted to know how much the GP and RL components contribute to the performance. Completely removing the RL component creates a stand-alone GP solver running an a random-restart like fashion. This alone does pretty well. It has been observed that that even though GP is not supposed to get stuck in local minima, bounded within a reasonable finite period of time, it does. Using random-restart has been observed to help in this case [Houck et al., 1996, Ghannadian et al.,

Table 3: Ablations are shown for a variety of different conditions. Percent recovery and 95% confidence interval is determined the same as with table 2. Top performing numbers are bolded.

	Mean % Recovery			
	All	Nguyen	R	Anonymous
Using PQT	**74.92**	92.33	**33.33**	**71.09**
No RL samples to train RNN (all *off-policy*)	74.81	93.00	29.33	**71.09**
Using VPG	74.27	**93.67**	28.00	70.00
RL Entropy Weight = 0	73.95	92.00	30.67	70.00
Using RSPG	73.95	92.67	39.33	69.82
(A) GP Mutate Uniform Only	72.65	91.67	25.33	68.73
(B) GP No inv/trig Constraint	71.35	91.33	32.00	65.82
(A) and *(B)*	68.11	91.33	21.33	61.82
No GP samples to train RNN (all *on-policy*)	66.27	89.00	24.00	59.64
(C) RL Learning Rate = 0	65.95	90.67	17.33	59.09
Random Restart *(GP Only)*	63.57	88.67	2.67	58.18
(D) No RL seeds to GP	63.24	89.00	2.67	57.45
(C) and *(D)*	63.14	88.67	2.67	57.45
RL Only [Petersen et al., 2021]	45.19	83.58	0.00	30.41
95% Confidence Interval ($n = 32$)	$\pm$ 1.543	$\pm$ 1.761	$\pm$ 2.809	$\pm$ 1.321

Table 4: Minimum normalized mean squared error (best expression found) from 25 different random number seeded runs of either DREGS or DSR on three very difficult to solve benchmarks is shown.

Benchmark	Minimum NMSE	
	GP-Meld	DSR
Nguyen-12	1.406E-04	2.014E-02
R1	3.547E-05	2.736E-03
R2	2.826E-04	5.313E-03

1996, Fukunaga, 2010]. If we replace RL but set the learning rate to zero, interestingly performance is enhance by a small but significant amount. The untrained RNN seems to supply better random population seeds than the more commonly used method [Fortin et al., 2012]. Having a learning policy-gradient RNN increases performance by 13.6%.

Method limitations. The primary limiting factor to our approach is that there are still some equations which cannot be completely recovered. While it may be a matter of finding the right set of hyperparameters, we have not as yet recovered Nguyen-12, R1 or R2. R3 can be recovered about 1 in 25 runs. Anonymous-7 and Anonymous-8 are recovered about once in every few hundred runs. Excluding R1*,R2*,R3* and Nguyen-12*, 86.49% of out benchmark expressions are easily recoverable. This is done by running 25 different random number seeded regression sessions in parallel. On our prior described workstation, this takes between 30 seconds and 17 minutes to complete. 91.89% of equations are recoverable if we are willing to wait hours rather than minutes. While we have not yet solved three benchmarks, table 4 shows that we still get closer than DSR.

6 Future Work and Conclusion

Our future work will include analysis of methods that might enhance the performance policy-gradient given that we are training from off-policy samples. We will also further examine why Nguyen-12, R1 and R2 seem intractable and see if we can come to a solution. We also plan on doing in-depth analysis of DREGS with noisy data. Things which are currently in the works are applications to explainable controller optimization (e.g. bipedal walker or lunar lander) and human infectious disease anti-body discovery.

We have developed a novel method to solve symbolic regression using policy-gradient reinforcement learning to find improved seeding for a generic programming solver that runs in a random-restart like manner. The results are state-of-the-art and outperform the best performing method by very comfortable margins.

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.

Sungsoo Ahn, Junsu Kim, Hankook Lee, and Jinwoo Shin. Guiding deep molecular optimization with genetic exploration. In *Proceeding of the Conference on Neural Information Processing Systems (NeurIPS)*, 2020.

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

Simyung Chang, John Yang, Jaeseok Choi, and Nojun Kwak. Genetic-gated networks for deep reinforcement learning. In *Proceeding of the Conference on Neural Information Processing Systems (NeurIPS)*, 2018.

Diqi Chen, Yizhou Wang, and Wen Gao. Combining a gradient-based method and an evolution strategy for multi-objective reinforcement learning. *Applied Intelligence*, 50, 2020a.

Qiong Chen, Mengxing Huang, Qiannan Xu, Hao Wang, and Jinghui Wang. Reinforcement learning-based genetic algorithm in optimizing multidimensional data discretization scheme. *Mathematical Problems in Engineering*, 2020b.

Yukang Chen, Gaofeng Meng, Qian Zhang, Shiming Xiang, Chang Huang, Lisen Mu, and Xinggang Wang. Renas: Reinforced evolutionary neural architecture search. In *Proceeding of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019.

Dario Floreano, Peter Dürr, and Claudio Mattiussi. Neuroevolution: from architectures to learning. *Evolutionary Intelligence*, 1:47–62, 2008.

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

Alex S. Fukunaga. Restart scheduling for genetic algorithms. In *Proceeding of the 5th International Conference on Parallel Problem Solving from Nature*, 2010.

Tanmay Gangwani and Jian Peng. Policy optimization by genetic distillation. In *Proceeding of the International Conference on Learning Representations (ICLR)*, 2018.

Farzad Ghannadian, Cecil Alford, and Ron Shonkwiler. Application of random restart to genetic algorithms. *Intelligent Systems*, 95:81–102, 1996.

Peter W Glynn and Donald L Iglehart. Importance sampling for stochastic simulations. *Management science*, 35(11):1367–1392, 1989.

Christopher R. Houck, Jeffrey A. Joines, and Michael G. Kay. Comparison of genetic algorithms, random restart, and two-opt switching for solving large location-allocation problems. *Computers and Operations Research*, 23:587–596, 1996.

James P Hughes, Stephen Rees, S Barrett Kalindjian, , and Karen L Philpott. Principles of early drug discovery. *British journal of pharmacology*, 162:1239—-1249, 2011.

Sam Ade Jacobs, Tim Moon, Kevin McLoughlin, Derek Jones, David Hysom, Dong H Ahn, John Gyllenhaal, Pythagoras Watson, Felice C Lightstone, Jonathan E Allen, Ian Karlin, and Brian Van Essen. Enabling rapid covid-19 small molecule drug design through scalable deep learning of generative models. *The International Journal of High Performance Computing Applications*, 2021.

Shauharda Khadka and Kagan Tumer. Evolution-guided policy gradient in reinforcement learning. In *Proceeding of the Conference on Neural Information Processing Systems (NeurIPS)*, 2018.

Sookyung Kim, Mikel Landajuela, Brenden K. Petersen, Claudio P. Santiago, Ruben Glatt, T. Nathan Mundhenk, Jacob F. Pettit, and Daniel M. Faissol. Discovering symbolic policies with deep reinforcement learning. In *Proceeding of the International Conference on Machine Learning (ICML)*, 2021.

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

Krzysztof Krawiec and Tomasz Pawlak. Approximating geometric crossover by semantic backpropagation. In *Proceeding of the fifteenth annual conference on Genetic and evolutionary computation (GECCO)*, 2013.

Andrey Kurenkov. Reinforcement learning's foundational flaw. *The Gradient*, 2018.

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

Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. In *Proceeding of the International Conference on Learning Representations (ICLR)*, 2016.

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

Benno Lüders, Mikkel Schläger, Aleksandra Korach, and Sebastian Risi. Continual and one-shot learning through neural networks with dynamic external memory. In *In European Conference on the Applications of Evolutionary Computation*, pages 886–901, 2017.

Risto Miikkulainen1, Jason Liang, Elliot Meyerson, Aditya Rawal, Dan Fink, OlivierFrancon, Bala Raju, Hormoz Shahrzad, Arshak Navruzyan, Nigel Duffy, and Babak Hodjat. Evolving deep neural networks. *Artificial Intelligence in the Age of Neural Networks and Brain Computing*, 2019.

Amanda J. Minnich, Kevin McLoughlin, Margaret Tse, Jason Deng, Andrew Weber, Neha Murad, Benjamin D. Madej, Bharath Ramsundar, Tom Rush, Stacie Calad-Thomson, Jim Brase, and Jonathan E. Allen. Ampl: A data-driven modeling pipeline for drug discovery. *Journal of Chemical Information and Modeling*, 60:1955–1968, 2020.

L. Montastruc, C. Azzaro-Pantel, L. Pibouleau, and S. Domenech. Use of genetic algorithms and gradient based optimization techniques for calcium phosphate precipitation. *Chemical Engineering and Processing: Process Intensification*, 43(10):1289–1298, 2004. ISSN 0255-2701. doi: https://doi.org/10.1016/j.cep.2003.12.002. URL `https://www.sciencedirect.com/science/article/pii/S0255270103002733`.

Brenden K Petersen, Mikel Landajuela, T Nathan Mundhenk, Claudio P Santiago, Soo K Kim, and Joanne T Kim. Deep symbolic regression: Recovering mathematical expressions from data via risk-seeking policy gradients. *Proceeding of the International Conference on Learning Representations (ICLR)*, 2021.

Mariya Popova, Olexandr Isayev, and Alexander Tropsha. Deep reinforcement learning for de novo drug design. *Science advances*, 4, 2018.

Aloïs Pourchot and Olivier Sigaud. Cem-rl: Combining evolutionary and gradient-based methods for policy search. In *Proceeding of the International Conference on Learning Representations (ICLR)*, 2019.

Esteban Real, Chen Liang, David R. So, and Quoc V. Leg. Automl-zero: Evolving machine learning algorithms from scratch. In *Proceeding of the International Conference on Machine Learning (ICML)*, 2019.

Sebastian Risi and Julian Togelius. Neuroevolution in games: State of the art and open challenges. *IEEE Transactions on Computational Intelligence and AI in Games*, 9:25–41, 2017.

Subham S Sahoo, Christoph H Lampert, and Georg Martius. Learning equations for extrapolation and control. In *Proceedings of the 35th International Conference on Machine Learning (ICML)*, 2018.

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

Gisbert Schneider. *De novo molecular design*. John Wiley & Sons, 2013.

Gisbert Schneider and Uli Fechner. Computer-based de novo design of drug-like molecules. *Nature Reviews Drug Discovery*, 4:649–663, 2005.

Adarsh Sehgal, Hung La, Sushil Louis, and Hai Nguyen. Deep reinforcement learning using genetic algorithm for parameter optimization. In *Proceeding of the IEEE International Conference on Robotic Computing (ICRC)*, 2019.

Kenneth O. Stanley and Risto Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary computation*, 10:99–127, 2002.

Kenneth O. Stanley, Jeff Clune, Joel Lehman1, and Risto Miikkulainen. Designing neural networks through neuroevolution. *Nature Machine Intelligence*, 1, 2019.

Felipe Petroski Such, Vashisht Madhavan, Edoardo Conti, Joel Lehman, Kenneth O. Stanley, and Jeff Clune. Deep neuroevolution: Genetic algorithms are a competitive alternative for training deep neural networks for reinforcement learning. In *arXiv:1712.06567*, 2018.

Yuan Tian, Qin Wang1, Zhiwu Huang, Wen Li, Dengxin Dai, Minghao Yang, Jun Wang, , and Olga Fink. Off-policy reinforcement learning for efficient and effective gan architecture search. In *Proceeding of the European Computer Vision Conference (ECCV)*, 2020.

Alexander Topchy and William Punch. Faster genetic programming based on local gradient search of numeric leaf values. In *Proceeding of the third annual conference on Genetic and evolutionary computation (GECCO)*, 2001.

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

Nguyen Quang Uy, Nguyen Xuan Hoai, Michael O'Neill, R.I. McKay, and Edgar Galvan-Lopez. Semantically-based crossover in genetic programming: Application to realvalued symbolic regression. *Genetic Programming and Evolvable Machines*, 12:91–119, 2014.

Daan Wierstra, Tom Schaul, Jan Peters, and Juergen Schmidhuber. Natural evolution strategies. In *Proceeding of the IEEE Congress on Evolutionary Computation*, 2008.

Cenqi Yan, Stephen Barlow, Zhaohui Wang, He Yan, Alex K-Y Jen, Seth R Marder, , and Xiaowei Zhan. Non-fullerene acceptors for organic solar cells. *Nature Reviews Materials*, 3:1—-19, 2018.

Mengjie Zhang and Will Smart. Genetic programming with gradient descent search for multiclass object classification. In Maarten Keijzer, Una-May O'Reilly, Simon Lucas, Ernesto Costa, and Terence Soule, editors, *Genetic Programming*, pages 399–408, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg. ISBN 978-3-540-24650-3.

7 Required NeurIPS Paper Checklist

1. For all authors...
   (a) Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope? [Yes] **It would be a bad idea not to do so.**
   (b) Did you describe the limitations of your work? [Yes] **As is customary.**
   (c) Did you discuss any potential negative societal impacts of your work? [N/A] **The paper does not pertain to any of the topics that would be associated with having a potential negative impact.**
   (d) Have you read the ethics review guidelines and ensured that your paper conforms to them? [Yes]
2. If you are including theoretical results...
   (a) Did you state the full set of assumptions of all theoretical results? [N/A]
   (b) Did you include complete proofs of all theoretical results? [N/A]
3. If you ran experiments...
   (a) Did you include the code, data, and instructions needed to reproduce the main experimental results (either in the supplemental material or as a URL)? [Yes] **The camera ready version will point to the usual git repo which will have everything.**
   (b) Did you specify all the training details (e.g., data splits, hyperparameters, how they were chosen)? [Yes] **As is expected**
   (c) Did you report error bars (e.g., with respect to the random seed after running experiments multiple times)? [Yes] **It's a good thing to do. Not many people seem to do it anyways.**
   (d) Did you include the total amount of compute and the type of resources used (e.g., type of GPUs, internal cluster, or cloud provider)? [Yes] **This is not quite as critical for our work, but we do include performance metrics and basic hardware descriptions.**
4. If you are using existing assets (e.g., code, data, models) or curating/releasing new assets...
   (a) If your work uses existing assets, did you cite the creators? [Yes] **Any source code used is of cource cited.**
   (b) Did you mention the license of the assets? [N/A]
   (c) Did you include any new assets either in the supplemental material or as a URL? [N/A]
   (d) Did you discuss whether and how consent was obtained from people whose data you're using/curating? [N/A]
   (e) Did you discuss whether the data you are using/curating contains personally identifiable information or offensive content? [N/A]
5. If you used crowdsourcing or conducted research with human subjects...
   (a) Did you include the full text of instructions given to participants and screenshots, if applicable? [N/A]
   (b) Did you describe any potential participant risks, with links to Institutional Review Board (IRB) approvals, if applicable? [N/A]
   (c) Did you include the estimated hourly wage paid to participants and the total amount spent on participant compensation? [N/A]