Using Denoising Autoencoder Genetic Programming to Control Exploration and Exploitation in Search

David Wittenberg(✉)

Johannes Gutenberg University, Mainz, Germany
wittenberg@uni-mainz.de

Abstract. Denoising Autoencoder Genetic Programming (DAE-GP) is a novel neural network-based estimation of distribution genetic programming (EDA-GP) algorithm that uses denoising autoencoder long short-term memory networks as a probabilistic model to replace the standard mutation and recombination operators of genetic programming (GP). At each generation, the idea is to flexibly identify promising properties of the parent population and to transfer these properties to the offspring where the DAE-GP uses denoising to make the model robust to noise that is present in the parent population. Denoising partially corrupts candidate solutions that are used as input to the model. The stronger the corruption, the stronger the generalization of the model. In this work, we study how corruption strength affects the exploration and exploitation behavior of the DAE-GP. For a generalization of the royal tree problem (high-locality problem), we find that the stronger the corruption, the stronger the exploration of the solution space. For the given problem, weak corruption resulting in a stronger exploitation of the solution space performs best. However, in more rugged fitness landscapes (low-locality problems), we expect that a stronger corruption resulting in a stronger exploration will be helpful. Choosing the right denoising strategy can therefore help to control the exploration and exploitation behavior in search, leading to an improved search quality.

Keywords: Genetic Programming · Estimation of Distribution Algorithms · Probabilistic Model-Building · Denoising Autoencoders

1 Introduction

Estimation of distribution genetic programming (EDA-GP) algorithms are meta-heuristics for variable-length combinatorial optimization problems that sample from a learned probabilistic model, replacing the standard mutation and recombination operators of genetic programming (GP). At each generation, the idea is to first learn the properties of promising candidate solutions of the parent population (*model building*) and to then sample from the model to transfer the learned properties to the offspring (*model sampling*) [9].

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2022
E. Medvet et al. (Eds.): EuroGP 2022, LNCS 13223, pp. 102–117, 2022.
https://doi.org/10.1007/978-3-031-02056-8_7

An example of an EDA-GP is denoising autoencoder genetic programming (DAE-GP) that uses denoising autoencoder long short-term memory networks (DAE-LSTMs) as a probabilistic model [25]. In comparison to previous EDA-GP approaches, it has the advantage that the model does not impose any assumptions about the relationships between problem variables which allows the DAE-GP to flexibly identify and model relevant properties of the parent population. The DAE-GP captures dependencies between problem variables by first encoding candidate solutions (in prefix notation) to the latent space and then reconstructing the candidate solutions from the latent space. For model building, the DAE-GP is trained to minimize the reconstruction error between the encoded and decoded candidate solutions. For model sampling, candidate solutions are propagated through the trained model to transfer the learned properties to the offspring [25].

The DAE-GP uses denoising to prevent the model from learning the simple identity function [25]. The idea is to partially corrupt input candidate solutions to make the model robust to noise that is present in the parent population. The stronger the corruption, the stronger the generalization of the model [24]. Previous work on estimation of distribution algorithms (EDA), where candidate solutions have a fixed length of size n, found that exploration and exploitation in search can be controlled by the strength of corruption [16]. Exploration increases the diversity of a population by introducing new candidate solutions into search; exploitation reduces diversity by focusing a population of candidate solutions on promising areas of the solution space [19]. Adjusting the corruption strength can therefore help to balance exploration and exploitation leading to a more successful search: we either increase diversity to overcome local optima avoiding premature convergence, or we decrease diversity to exploit promising solution spaces [16].

In this work, we study how corruption strength affects the exploration and exploitation behavior of the DAE-GP. Wittenberg et al. [25] used subtree mutation to corrupt input candidate solutions. Subtree mutation randomly selects a node in a tree and replaces the subtree at that node with a new random subtree generated by ramped half-and-half. The use of subtree mutation has the advantage that it leads to a variation in tree size (the number of nodes in parse tree) [25]. However, applying subtree mutation complicates the control of corruption strength: as subtree mutation randomly selects a subtree to be replaced by a new random subtree, corruption is stronger if the root of the selected subtree is nearer to the root of the parse tree. Furthermore, increasing or decreasing corruption strength is difficult.

Therefore, this paper introduces Levenshtein edit as a new and improved denoising strategy. Levenshtein edit is based on the Levenshtein distance [12] and operates on the string representation of a candidate solution (prefix expression). It uses insertion (add one node), deletion (remove one node), and substitution (replace one node by another node) as edit operators to corrupt a candidate solution. The advantage of using Levenshtein edit over subtree mutation is that we can accurately adjust corruption strength. The more nodes we edit, the stronger the corruption, and the more we force the DAE-GP to focus on general properties of the parent population.

We compare the performance of the DAE-GP with Levenshtein edit and different levels of corruption strength (2%, 5%, 10%, 20%) to a DAE-GP with subtree mutation and standard GP, and analyze the impact of corruption strength on search. We find that corruption strength strongly influences both the performance and the exploration and exploitation behavior of the DAE-GP: the stronger we corrupt input candidate solutions, the stronger the exploration. However, exploration is useful, only if we want to escape from local optima. For the generalization of the royal tree problem (which is an easy problem with high locality), we find that the DAE-GP with weak corruption (Levenshtein edit with 5% corruption strength) performs best. However, when facing more rugged fitness landscapes, a stronger degree of exploration can be helpful. We therefore believe that the denoising strategy is the key to the success of the DAE-GP: it allows us to control the level of exploration and exploitation in search helping us to improve search quality.

In Sect. 2, we present related work on EDA-GP. We describe DAE-LSTMs in Sect. 3, where we focus on the architecture, the denoising strategy, and on model building and sampling. In Sect. 4, we introduce the experiments and discuss the results. We draw conclusions in Sect. 5.

2 Related Work

We can categorize research on EDA-GP into two research streams [9,21]: The first one uses probabilistic prototype trees (PPT) as a model. Given the maximum arity a of the functions in the function set (the interior nodes of a GP parse tree), a PPT is a full tree of arity a where we set the depth of the PPT equal to the maximum tree depth d_{max}. At each node of the PPT, the idea is to first build a multinomial probability distribution over the set of allowed functions (internal nodes) and terminals (leaf nodes) and to then update the distributions according the candidate solutions that are presented to the model. In 1997, Salustowicz and Schmidhuber [20] introduced PPTs as the first probabilistic model in EDA-GP called probabilistic incremental program evolution (PIPE) [20]. Based on PIPE that evolves univariate probability distributions, EDA-GP models have been developed that capture dependencies between nodes in a PPT tree. Examples are the bivariate estimation of distribution programming (EDP) [28] or the multivariate program optimization with linkage estimation (POLE) [4,6]. Hasegawa and Iba [6] report that POLE needs less fitness evaluations than standard GP to solve the MAX, the deceptive MAX, and the royal tree problem [6].

The second stream of research uses grammars as EDA-GP model [9,21]. Here, Ratle and Sebag [18] presented stochastic grammar-based genetic programming (SG-GP) as the first grammar-based approach in 2001. SG-GP uses stochastic context-free grammar (SCFG) as a probabilistic model. The idea is to first identify a set of production rules for a problem with weights attached to the production rules and to then update these weights according to usage counts of the production rules in a parent population [18]. Since SG-GP assumes the production rules to be independent, more sophisticated EDA-GP models capturing more complex grammars have been developed. Consequently, program

with annotated grammar estimation (PAGE) is an extension that uses expectation maximization (EM) or variational Bayes (VB) to learn production rules with latent annotations. A latent annotation can be, e.g., the position or the depth of a node in a tree [5]. Another extension is grammar-based genetic programming with a Bayesian network (BGBGP) that was introduced by Wong et al. [26] in 2014. BGBGP uses Bayesian networks with stochastic context-sensitive grammars (SCSG) as a model. Compared to SCFG, SCSG additionally incorporate contextual information allowing the Bayesian network to learn dependencies between production rules [26]. To further refine the BGBGP, Wong et al. [27] added (fitness) class labels to the model. The authors argue that this allows the model to differentiate between good and poor candidate solutions helping the model to find better solutions. For the deceptive MAX and the asymmetric royal tree problem, the model outperforms POLE, PAGE-EM, PAGE-VB, and grammar-based GP in the number of fitness evaluations [27].

One example of an EDA-GP model that does not rely on PPTs or grammars is the n-gram GP proposed by Poli and McPhee [14], where n-grams are used to model relationships between a group of n consecutive sequences of instructions that can learn dependencies in linear GP. Similarly, Hemberg et al. [7] suggested operator free genetic programming (OFGP), which learns n-grams of ancestor node chains. An n-gram of ancestors is the sequence of a node and its n-1 ancestor nodes in a GP parse tree. However, for the Pagie-2D problem, OFGP could not outperform standard GP [7].

Wittenberg et al. [25] recently suggested DAE-GP that uses denoising autoencoder long short-term memory networks (DAE-LSTMs) as a probabilistic model. For a generalization of the royal tree problem, the DAE-GP outperforms standard GP. The DAE-GP can better identify promising areas of the solution space compared to standard GP resulting in a more efficient search in the number of fitness evaluations, especially in large search spaces [25]. The authors argue that, compared to previous EDA-GP approaches, the flexible model representation is the key reason for the high performance, allowing the model to identify in parallel, both position as well as context of relevant substructures [25].

The idea of using DAE as probabilistic models in EDA has earlier been presented by Probst [15] who introduced DAE-EDA. DAE-EDA was designed for problems where candidate solutions follow a fixed-length representation [15]. For the NK landscapes, deceptive traps and HIFF problem, Probst and Rothlauf [16] show that the DAE-EDA yields competitive results compared to the Bayesian optimization algorithm (BOA). However, DAE-EDA is better parallelizable, making it the preferred choice especially in large search spaces. Furthermore, the authors show that corruption strength has a strong impact on exploration and exploitation in search. Adjusting the level of corruption can therefore help to either increase exploration which helps to overcome local optima, or to exploit relevant solution spaces making search more efficient [16].

3 Denoising Autoencoder LSTMs

DAE-LSTMs are artificial neural networks that consist of an encoding and a decoding LSTM: the encoding LSTM encodes a candidate solution (a linear sequence in prefix expression) to the latent space; the decoding LSTM decodes the latent space back to a candidate solution. Since we train the DAE-LSTM to reconstruct the input, the architecture is also referred to as autoencoder long short-term memory network (AE-LSTM) [22], where we use denoising on input candidate solutions to prevent the model from learning the simple identity function. Denoising transforms the AE-LSTM into a DAE-LSTM. When using DAE-LSTMs as a probabilistic model in EDA-GP (DAE-GP), we repeat the following two steps at each generation: first, we train the model to learn relevant properties of our parent population (model building). Then, we propagate candidate solutions through the trained DAE-LSTM to transfer the learned properties to the offspring (model sampling).

In the following sections, we first explain the architecture of AE-LSTMs and the concept of denoising, where we introduce Levenshtein edit as a new denoising strategy. Then, we describe the training as well as the sampling procedure where syntax control is used to restrict the sample space to syntactically valid candidate solutions.

3.1 Autoencoder LSTMs

Figure 1 shows the architecture of an AE-LSTM with one input layer, one hidden layer (consisting of LSTM memory cells), and one output layer. It is based on the architecture presented in [25]. x and o represent the input and output candidate solution of length m and k, respectively. h is the hidden state at time step t, where the total number of time steps corresponds to $T = m + k$ ($m, k \in \mathbb{N}$). The encoding LSTM (left) first sequentially encodes a candidate solution x, with x_t, $t \in \{1, 2, .., m\}$ through the encoding function $g(x)$, where each x_t represents a function or terminal of a candidate solution in our parent population. At each time step t (except $t = 0$), the LSTM memory cell then receives three inputs: the current input x_t, the previous hidden state h_{t-1} and the previous cell state c_{t-1} (not shown here). The idea of transferring information from one time step to the next is to capture long-term dependencies in training data [8]. After complete processing of the input candidate solution x, we copy h_m and c_m, and transfer it to the decoding LSTM, thus $h_{m+1} = h_m$ and $c_{m+1} = c_m$. The decoding LSTM (right) then uses the decoding function $d(h)$ and decodes h_t back to an output candidate solution o, with the aim to reconstruct the input candidate solution x. Using o_t as input in o_{t+1} helps to further reduce the reconstruction error [22]. Similar to [22] and [25], we reverse the input candidate solution x to allow the model to learn low range correlations in training data.

3.2 Suggesting a New Denoising Strategy: Levenshtein Edit

The aim of the AE-LSTM is to reconstruct the input. Given that the hidden layer is sufficiently large, a trivial way to solve this task is to learn the simple

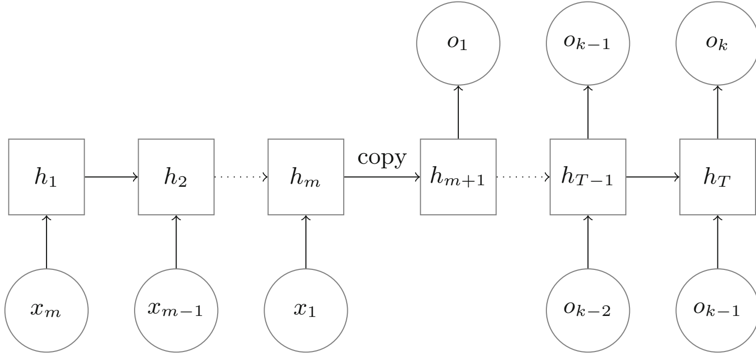


Fig. 1. Autoencoder LSTM assuming one hidden layer

identity function, which means that the AE-LSTM simply replicates the candidate solutions given as input. Since we want to learn a more useful representation of the properties of our parent population, we apply denoising on input candidate solutions, transforming the AE-LSTM into a DAE-LSTM. Based on the first DAE presented by Vincent et al. [24] in 2008, the idea is to partially corrupt input candidate solutions making the model robust to noise that is present in our parent population.

At each generation g, we use the corruption function $c(x)$ to denoise the candidate solutions that were previously selected as promising candidate solutions from population P_g. We can formally describe the process by

$$\tilde{x}^i = c(x^i) \;\; \forall i \in \{1, .., N\}, \tag{1}$$

where $\tilde{x}^i$ is the corrupted version of the ith candidate solution x in the training set X (of size N) [25].

As a new corruption function $c(x)$, we introduce Levenshtein edit. Levenshtein edit operates on the string representation of x (prefix expression) and uses insertion (add one node), deletion (remove one node), and substitution (replace one node by another node) to transform x into $\tilde{x}$. We control the corruption strength by a priori defining a corruption percentage p $(0 < p < 1)$. Given a function set F, a terminal set T, and a candidate solution x, with x_j, $j \in \{1, 2, .., m\}$, we corrupt x by iteratively processing each node x_j, where each x_j has a chance of p to be corrupted: with uniform probability, we either insert a random symbol $s \in F \cup T$ at index j (insertion), we delete x_j (deletion), or we delete x_j *and* insert a random symbol $s \in F \cup T$ at index j (substitution). Note that these edit operations may produce corrupted candidate solutions $\tilde{x}$ that do not follow GP syntax. However, sampling with syntax control (see Sect. 3.4) ensures that output candidate solutions o are syntactically valid. Using Levenshtein edit as denoising strategy has several advantages: similar to subtree mutation presented in [25], we introduce variance in tree size. This is desirable since it introduces additional variation into $\tilde{x}$. However, this variation should

not lead to a bias towards larger or smaller trees. When using subtree mutation as denoising strategy, we randomly select a subtree to be replaced. Depending on the size of the selected subtree, we easily corrupt larger or smaller parts of x resulting in a bias in tree size. The situation is different for Levenshtein edit: here, we randomly choose denoising operators that iteratively either increase (insertion), decrease (deletion), or maintain (substitution) the size of x. Thus, for any p, the expected tree size of $\tilde{x}$ is equal to the tree size of x, which means that we are able to introduce variation without inducing a bias in tree size. Furthermore, we can easily control the corruption strength by adjusting p. The larger p, the stronger the variation, and the stronger the corruption. The results in Sect. 4 will show that this helps to control exploration and exploitation in search.

3.3 Training Procedure

At each generation g, we train a DAE-LSTM (from scratch) according to the training procedure shown in Algorithm 1. It is similar to the training procedure presented in [25]. We first initialize the trainable parameters θ of our network, where $W^{'}$, $b^{'}$, and $W^{''}$, $b^{''}$ (Algorithm 1, line 1) denote the trainable weights and biases of the encoding and decoding LSTM, respectively. Then, we iteratively adjust the values of the trainable parameters θ using gradient descent. Given the corruption percentage p, we first transform the candidate solution x^i into $\tilde{x}^i$ (Algorithm 1, line 4). Then, we propagate $\tilde{x}^i$ through the DAE-LSTM, using the encoding function $g(x)$ (Algorithm 1, line 5) and the decoding function $d(x)$ (Algorithm 1, line 6). We compute the reconstruction error using the multiclass cross entropy loss function by

$$\theta := \min_{\theta} \sum_{i=1}^{N} Err(x^i, o^i), \tag{2}$$

where o^i is the output candidate solution and x^i the original (not the corrupted) input candidate solution. We update the parameters θ into the direction of the negative gradient and control the strength of the update using the learning rate α $(0 < \alpha < 1)$ (Algorithm 1, line 7).

Algorithm 1. Pseudocode for training a DAE-LSTM

1: **Initialize** $\theta = \{W^{'}, b^{'}, W^{''}, b^{''}\}$
2: **while** not converged **do**
3:     **for each** candidate solution x^i in training set X **do**
4:         $\tilde{x}^i = c(x^i; p)$
5:         $h = g(\tilde{x}^i; \theta)$
6:         $o^i = d(h; \theta)$
7:         $\theta := \theta - \alpha * \frac{\partial Err(x^i, o^i)}{\partial \theta}$
8:     **end for**
9: **end while**

We use early stopping to prevent the DAE-LSTM from overfitting. Given a hold-out validation set U, we stop training as soon as the validation error $Err(x^j, o^j)$, with $x^j, o^j \in U$, converges. We measure error convergence by observing the number of epochs that the validation error does not improve. As soon as we reach 200 epochs of no improvement, we stop training and use those parameters θ for sampling that minimize the validation error.

3.4 Sampling with Syntax Control

We use the DAE-LSTM with the trained parameters θ to sample new candidate solutions o forming the offspring population P_{g+1}. The procedure is shown in Algorithm 2 and based on [1,16,25]. Given θ (Algorithm 2, line 1), we first randomly pick a candidate solution x of our training set X (Algorithm 2, line 2). Then, we corrupt x into $\tilde{x}$ (Algorithm 2, line 3) using the same denoising strategy as during training and propagate $\tilde{x}$ through the DAE-LSTM (Algorithm 2, lines 4–5), where we add the resulting output candidate solution o to P_{g+1} (Algorithm 2, line 6).

Algorithm 2. Pseudocode for sampling from a DAE-LSTM

1: **Given** the trained DAE-LSTM with $\theta = \{W^{'}, b^{'}, W^{''}, b^{''}\}$
2: **Pick** $x \in X$ randomly
3: $\tilde{x} = c(x, p)$
4: $h = g(\tilde{x}; \theta)$
5: $o = d(h; \theta)$
6: **Add** o to new population P_{g+1}

Furthermore, we introduce a syntax control mechanism that only allows syntactically valid candidate solutions to be sampled. The mechanism proceeds as follows: at each time step t, with $t \in \{m+1, m+2, .., T\}$, when decoding h back to o (Algorithm 2, line 5), the DAE-LSTM generates a probability distribution q over the set of functions and terminals (defined by F and T). Similar to grow initialization [11], we first identify the set of functions and terminals that generate a syntactically valid candidate solution. Then, we set the classes of invalid functions and terminals in q to zero and normalize the remaining probabilities in q back to one, where we use the updated probability distribution to sample o_t.

Without denoising, syntax control is usually not needed since the complexity of the DAE-LSTM is sufficient to also learn correct syntax. However, the stronger the corruption, the more difficult it becomes for the DAE-LSTM to sample syntactically valid candidate solutions, since corrupted candidate solutions used as input to the model no longer belong to the same parent population as X. In these cases, syntax control is very useful: we prevent the DAE-LSTM from inefficient resampling and allow the model to explore new solution spaces, which can help to overcome local optima and to avoid premature convergence.

4 Experiments

We present the experimental setup for studying the influence of denoising on search. We find that the DAE-GP with Levenshtein edit and $p = 0.05$ outperforms a DAE-GP with subtree mutation and standard GP. Furthermore, we show that corruption strength p strongly affects search: the stronger the corruption, the stronger the exploration. Adjusting the corruption strength can therefore help to either exploit or explore relevant areas of the solution space.

4.1 Experimental Setup

For our study, we use the generalization of the royal tree problem presented in [25] as test problem. It is based on the royal tree problem introduced by Punch et al. [17] but uses the initialization method ramped half-and-half [11] to generate target candidate solutions x_{opt}. The idea is to define a fitness based on the structure of a candidate solution x by

$$fitness_x = \frac{lev(x, x_{opt})}{max(l_x, l_{x_{opt}})}, \tag{3}$$

where lev is the minimum Levenshtein distance, defined by the minimum number of insertion, deletion, and substitution operations necessary to transform x into x_{opt} [12]. Similar to [25], we divide lev by the maximum size l of x and x_{opt}, resulting in $fitness_x \in [0, 1]$: the closer x to x_{opt}, the better the fitness, where $fitness_x = 0$ means that x is identical to x_{opt} [25]. We tune the complexity of the problem by adjusting the minimum and maximum tree depths d_{min} and d_{max}, respectively. The larger the solution space, the more difficult the problem.

We implemented the experiments in Python using the evolutionary framework DEAP [3] and the neural network framework Keras [2]. Table 1 shows the GP and DAE-GP parameters. We use the Pagie-1 [13] function and terminal set and define two different problem settings, where we fix the minimum tree depth to $d_{min} = 3$ and set the maximum tree depth to $d_{max} \in \{4, 5\}$. We choose a population size of 500, use binary tournament selection, and run the experiments for a total of 100 generations. We use ramped half-and-half to generate both the initial population and the target candidate solutions x_{opt} and define 30 different x_{opt} per problem setting. Performing 5 runs per x_{opt} results in a total number of 150 runs that we aggregate per problem setting and algorithm. Since we consider six different algorithm configurations, we conduct 1,800 runs in total.

For GP, we follow the recommendations of Koza [11] and use subtree crossover as variation operator where we set an internal node bias to assure that 90% of the crossover points are functions. For the DAE-GP, we have to a priori define a set of hyperparameters. Note that we did not conduct a hyperparameter optimization. We set the number of hidden layers to one and the number of hidden neurons equal the maximum size l of the candidate solutions used as input to the model. We found that the complexity of the model is sufficient to learn complex relationships in training data while allowing efficient model building

Table 1. GP and DAE-GP Parameters

Parameter	Setting
function set	$F = \{+, -, *, /, sin, cos, exp, log\}$
terminal set	$T = \{x, y, 1\}$
target cand. solutions	30 per problem setting
runs	5 per target candidate solution
population size	500
generations	100
initialization	ramped half-and-half
selection	tournament selection of size 2
tree depths	$d_{min} = 3$ and $d_{max} \in \{4, 5\}$
variation operator	GP: subtree crossover with internal node bias (90% functions, 10% terminals), DAE-LSTM: model building and sampling with Levenshtein edit using $p \in \{0.02, 0.05, 0.1, 0.2\}$ and subtree mutation using $d_{min}, d_{max} = 2$ [25]

and sampling. We split the parent population into 50% training set X and 50% validation set U, and set the batch size to 25 (10% of X). We use a learning rate of $\alpha = 0.001$ and perform adaptive moment estimation (Adam) [10] for gradient descent optimization. To study the impact of denoising on search, we vary the denoising strategy throughout the experiments: we test Levenshtein edit, with $p \in \{0.02, 0.05, 0.1, 0.2\}$, and a DAE-GP using subtree mutation, where previous work recommends to set the depth of the new subtree to $d_{min}, d_{max} = 2$ [25].

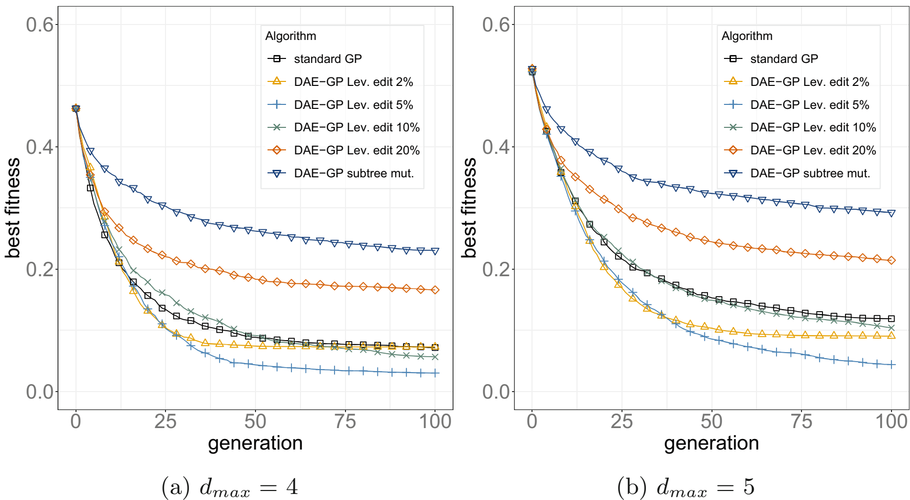
4.2 Performance Results

We first study the algorithm success rates for the two problem complexities ($d_{max} \in \{4, 5\}$) and the six different algorithm configurations. A run is successful as soon as the algorithm finds a candidate solution x during search that is identical to the target candidate solution x_{opt} ($fitness_x = 0$). Table 2 shows the average success rates after 100 generations. Each success rate represents the average over 150 runs (5 runs for each of the 30 target candidate solutions x_{opt}). As expected, the average success rates are higher for $d_{max} = 4$ compared to $d_{max} = 5$: the solution space becomes larger when choosing larger tree depths making it harder to find x_{opt}. However, the success rates differ strongly depending on the algorithm considered. For both $d_{max} = 4$ and $d_{max} = 5$, the DAE-GP with Levenshtein edit and $p = 0.05$ performs best, with an average success rate of 72.67% and 58.67%, respectively. Interestingly, increasing or decreasing p results in a loss in search success. While the DAE-GP with $p = 0.02$ and $p = 0.1$ yields similar average success rates compared to standard GP (51.33% vs. 59.33% vs. 50.00% for $d_{max} = 4$ and 38.00% vs. 35.33% vs. 36.67% for $d_{max} = 5$), we achieve low success rates using strong corruption: for $d_{max} = 4$ and $d_{max} = 5$, the DAE-GP with $p = 0.2$ only finds 26.00% and 16.67% of the target solutions, respectively. The DAE-GP with subtree mutation performs worst, with average

success rates of 16.00% and 1.33%, respectively. The high performance of the DAE-GP with Levenshtein edit and $p = 0.05$ indicates that the model successfully identifies and models relevant properties of the parent population and is able to transfer these properties to the offspring. However, performance strongly depends on the denoising strategy applied.

Table 2. Average success rates after 100 generations

Algorithm	$d_{max} = 4$	$d_{max} = 5$
Standard GP	50.00%	36.67%
DAE-GP Levenshtein edit $p = 2\%$	51.33%	38.00%
DAE-GP Levenshtein edit $p = 5\%$	72.67%	58.67%
DAE-GP Levenshtein edit $p = 10\%$	59.33%	35.33%
DAE-GP Levenshtein edit $p = 20\%$	26.00%	16.67%
DAE-GP subtree mutation	16.00%	1.33%



(a) $d_{max} = 4$ (b) $d_{max} = 5$

Fig. 2. Average best fitness over number of generations for problems of varying complexity.

Figure 2 plots the average best fitness over the number of generations. Since we face a minimization problem, we observe a general decrease in the average best fitness over the number of generations. The solution space is larger for $d_{max} = 5$, resulting in a best fitness level that is slightly higher compared to $d_{max} = 4$. Again, for both problem settings, the DAE-GP with Levenshtein edit and $p = 0.05$ performs best, confirming the results from Table 2. Interestingly,

in early generations, the DAE-GP with $p = 0.02$ finds similar best fitness candidate solutions compared to $p = 0.05$ but then hardly improves from generation $g = 30$ ($d_{max} = 4$) and $g = 40$ ($d_{max} = 5$), indicating that the algorithm has already converged. In contrast, when setting the corruption strength to $p = 0.1$, we observe a similar best fitness slope of the DAE-GP and standard GP, demonstrating a similar search behavior. When using $p = 0.2$ or subtree mutation as denoising strategy, the performance is much worse.

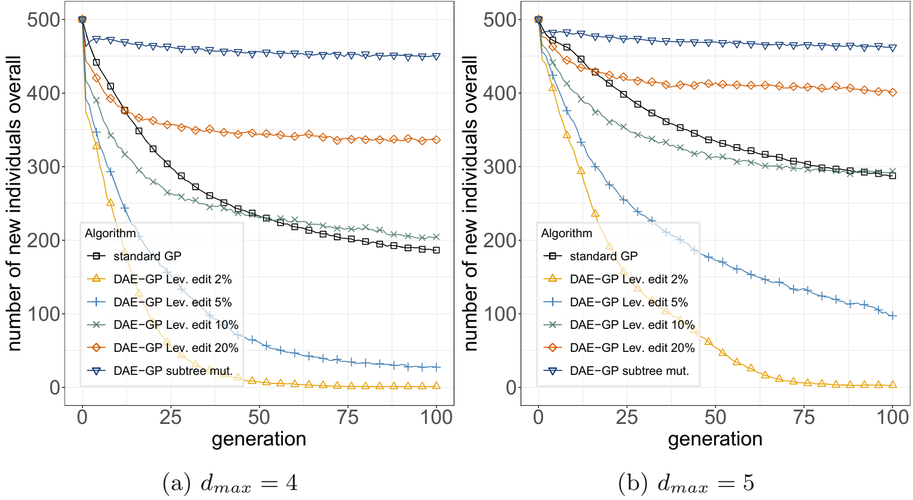
Given the distribution of the best fitness at the end of each run (generation 100), we conduct several (pairwise) Mann-Whitney U-Tests to test the hypothesis that the best fitness distributions are from the same population. Assuming a significance level of 0.05, we find that the DAE-GP with Levenshtein edit and $p = 0.05$ yields p-values <0.01 for all pairwise comparisons. The results indicate that this DAE-GP is significantly better than all other tested algorithms. When using $p = 0.1$ and comparing the DAE-GP to standard GP, we find p-values of 0.09 ($d_{max} = 4$) and 0.58 ($d_{max} = 5$). Similarly, when setting the corruption strength to $p = 0.02$ and comparing the DAE-GP to standard GP, we find p-values of 0.98 ($d_{max} = 4$) and 0.2 ($d_{max} = 5$). In both cases, the results indicate that the best fitness distributions do not significantly differ from each other, confirming the observation that these algorithms generate similar best fitness candidate solutions.

4.3 The Influence of Denoising on Search

The results above demonstrate that denoising has a strong impact on the performance of the DAE-GP. To better understand the influence of denoising on search, we study the exploration and exploitation behavior of the algorithms. Similar to [25], we approximate exploration and exploitation by examining the number of new candidate solutions over generations that have never been sampled before. Exploitation is stronger, if search introduces a lower number of new candidate solutions during search. In contrast, the more new candidate solutions we introduce into search, the stronger the exploration. According to Rothlauf [19], we need to find an appropriate and problem-specific balance between exploration and exploitation in search. For problems, where small variations on the genotype lead to small variations in fitness (*high-locality problems*), we usually need much less exploration compared to problems, where the fitness landscape is rugged (*low-locality problems*). Thus, depending on the problem at hand, we either need to increase exploitation, making search more efficient, or we need to increase exploration, helping search to keep diversity high and allowing to overcome local optima and to avoid premature convergence [19].

For the generalization of the royal tree problem, we plot results in Fig. 3. As expected, for both variants, we observe a general decrease in the number of candidate solutions over generations. Furthermore, the level of exploration is in general higher for $d_{max} = 5$, again because we face a larger solution space compared to $d_{max} = 4$.

When comparing different denoising strategies with each other, we notice that the level of exploration and exploitation strongly differs throughout the



(a) $d_{max} = 4$

(b) $d_{max} = 5$

Fig. 3. Mean number of new candidate solutions over number of generations for problems of varying complexity.

search. For Levenshtein edit, we observe that the larger the corruption strength p, the stronger the exploration. While the DAE-GP with $p = 0.02$ strongly decreases and converges towards zero at generation $g = 50$ ($d_{max} = 4$) and $g = 75$ ($d_{max} = 5$), setting corruption strength to $p = 0.2$ results in a strong exploration of the solution space. Interestingly, as noticed in Sect. 4.2, both settings lead to an inferior performance compared to $p = 0.05$. The DAE-GP with $p = 0.02$ easily gets stuck in local optima (premature convergence) as we tend to replicate the candidate solutions given as input. In contrast, the DAE-GP with corruption strength of $p = 0.2$ introduces too many new candidate solutions, resulting in an inefficient search. Thus, by setting corruption strength to $p = 0.05$, we find a good balance between exploration and exploitation in search.

Another interesting observation is that the level of exploration of standard GP is similar to the one of the DAE-GP, using Levenshtein edit with $p = 0.1$. Thus, we can adjust the corruption strength in a way that allows us to imitate the exploration and exploitation behavior of standard GP, also yielding similar performance results.

For subtree mutation as denoising strategy, we notice that the level of exploration is highest throughout the search, yielding the worst results. We think that the introduction of syntax control (see Sect. 3.4) is the main reason for the bad performance of subtree mutation compared to the results published in [25]. Syntax control allows the DAE-GP to introduce more new candidate solutions into search, which can be helpful to overcome local optima. However, for the generalization of the royal tree problem (high-locality problem), the strong exploration

leads to inferior performance. Instead, search with strong exploitation, as shown for Levenshtein edit $p = 0.05$, is more successful.

The results indicate that the denoising strategy is key to the success of the DAE-GP. It strongly influences exploration and exploitation in search and therefore affects performance. Thus, we believe that the denoising strategy should be adjusted depending on the problem at hand: while weaker corruption helps to improve search quality for high-locality problems, we expect stronger corruption to be more successful when we face rugged fitness landscapes (low-locality problems). Here, a stronger exploration of the solution space can help to overcome local optima and to avoid premature convergence.

5 Conclusions

The DAE-GP is an EDA-GP model based on artificial neural networks that flexibly identifies and models hidden relationships in training data. It uses denoising on input candidate solutions to make the model robust to noise that is present in the parent population. This paper introduced Levenshtein edit as a new and improved denoising strategy, allowing us to precisely control corruption strength. Furthermore, we implemented a new syntax control mechanism for sampling from the DAE-GP, allowing a higher level of exploration throughout the search.

We find that denoising strongly influences exploration and exploitation in search and therefore affects performance. The stronger we denoise input candidate solutions, the stronger the exploration. Exploration is especially useful for low-locality problems where we want to escape from local optima. In contrast, for high-locality problems, such as the generalization of the royal tree problem considered in this work, stronger exploitation is needed. Therefore the DAE-GP with low corruption strength (5%) performs best. The results show that the denoising strategy is key to the success of the DAE-GP: it permits us to control the exploration and exploitation behavior in search leading to an improved search quality.

In future work, we investigate the influence of denoising on other problem domains. We will study if we can dynamically control corruption strength throughout search. In addition, we think that Levenshtein edit as denoising strategy can still be improved. The denoising strategy presented in this paper operates on the string of a candidate solution, which easily destroys GP syntax. Thus, Levenshtein edit operating on a parse tree could be a promising approach. Furthermore, a hyperparameter optimization could further improve model quality, as well as other architectures, such as the transformer architecture [23]. Besides this, future work should investigate if a pre-training of the model before evolution helps to improve search quality.

Acknowledgements. I thank my team in Mainz, especially Franz Rothlauf, for insightful discussions on this topic, as well as Dirk Schweim and Malte Probst for previous work on this topic.

References

1. Bengio, Y., Yao, L., Alain, G., Vincent, P.: Generalized denoising auto-encoders as generative models. In: Advances on Neural Information Processing Systems (NIPS 2013), vol. 26, pp. 899–907 (2013)
2. Chollet, F.: keras. https://github.com/fchollet/keras (2015)
3. Fortin, F.A., De Rainville, F.M., Gardner, M.A., Parizeau, M., Gagńe, C.: DEAP: evolutionary algorithms made easy. J. Mach. Learn. Res. **13**(1), 2171–2175 (2012)
4. Hasegawa, Y., Iba, H.: Estimation of Bayesian network for program generation. In: Proceedings of the Third Asian-Pacific Workshop on Genetic Programming, pp. 35–46. Hanoi, Vietnam (2006)
5. Hasegawa, Y., Iba, H.: Estimation of distribution algorithm based on probabilistic grammar with latent annotations. In: Proceedings of the IEEE Congress on Evolutionary Computation, CEC 2007, pp. 1043–1050. IEEE (2007), https://doi.org/10.1109/CEC.2007.4424585
6. Hasegawa, Y., Iba, H.: A Bayesian network approach to program generation. IEEE Trans. Evol. Comput. **12**(6), 750–764 (2008). https://doi.org/10.1109/tevc.2008.915999
7. Hemberg, E., Veeramachaneni, K., McDermott, J., Berzan, C., O'Reilly, U.M.: An investigation of local patterns for estimation of distribution genetic programming. In: Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2012), pp. 767–774. ACM, Philadelphia (2012). https://doi.org/10.1145/2330163.2330270
8. Hochreiter, S., Schmidhuber, J.: Long short-term memory. Neural Comput. **9**(8), 1735–1780 (1997). https://doi.org/10.1162/neco.1997.9.8.1735
9. Kim, K., Shan, Y., Nguyen, X.H., McKay, R.I.: Probabilistic model building in genetic programming: a critical review. Gene. Program. Evol. Mach. **15**(2), 115–167 (2013). https://doi.org/10.1007/s10710-013-9205-x
10. Kingma, D.P., Ba, J.: Adam: a method for stochastic optimization. In: International Conference on Learning Representations. San Diego (2015)
11. Koza, J.R.: Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press, London (1992)
12. Kruskal, J.B.: An overview of sequence comparison: time warps, string edits, and macromolecules. Soc. Ind. Appl. Math. (SIAM) Rev. **25**(2), 201–237 (1983). https://doi.org/10.1137/1025045
13. Pagie, L., Hogeweg, P.: Evolutionary consequences of coevolving targets. Evol. Comput. **5**(4), 401–418 (1997)
14. Poli, R., McPhee, N.F.: A linear estimation-of-distribution GP system. In: Proceedings of the 11th European Conference on Genetic Programming (EuroGP 2008), pp. 206–217. Springer, Neapel (2008). https://doi.org/10.1007/978-3-540-78671-9
15. Probst, M.: Denoising autoencoders for fast combinatorial black box optimization. In: Proceedings of the Companion Publication of the Annual Conference on Genetic and Evolutionary Computation, pp. 1459–1460. ACM, Madrid (2015)
16. Probst, M., Rothlauf, F.: Harmless overfitting: Using denoising autoencoders in estimation of distribution algorithms. J. Mach. Learn. Res. **21**(78), 1–31 (2020). http://jmlr.org/papers/v21/16-543.html
17. Punch, B., Zongker, D., Goodman, E.: The royal tree problem, a benchmark for single and multi-population genetic programming. In: Angeline, P.J., Kinnear, K.E., Jr. (eds.) Advances in Genetic Programming II, pp. 299–316. MIT Press, Cambridge (1996)

18. Ratle, A., Sebag, M.: Avoiding the bloat with stochastic grammar-based genetic programming. In: Collet, P., Fonlupt, C., Hao, J.-K., Lutton, E., Schoenauer, M. (eds.) EA 2001. LNCS, vol. 2310, pp. 255–266. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-46033-0_21
19. Rothlauf, F.: Design of Modern Heuristics: Principles and Application, 1st edn. Springer, Berlin (2011). https://doi.org/10.1007/978-3-540-72962-4
20. Salustowicz, R., Schmidhuber, J.: Probabilistic incremental program evolution. Evol. Comput. **5**(2), 123–141 (1997). https://doi.org/10.1162/evco.1997.5.2.123
21. Shan, Y., McKay, R., Essam, D., Abbass, H.: A survey of probabilistic model building genetic programming. In: Pelikan, M., Sastry, K., CantúPaz, E. (eds.) Scalable Optimization via Probabilistic Modeling, pp. 121–160. Springer, Berlin (2006). https://doi.org/10.1007/978-3-540-34954-9
22. Srivastava, N., Mansimov, E., Salakhutdinov, R.: Unsupervised learning of video representations using LSTMs. In: Proceedings of the 32nd International Conference on Machine Learning (ICML 2015), pp. 843–852. ACM, Lille (2015). https://doi.org/10.5555/3045118.3045209
23. Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, Ł, Polosukhin, I.: Attention is all you need. Adv. Neural Inf. Process. Syst. **30**, 5998–6008 (2017)
24. Vincent, P., Larochelle, H., Bengio, Y., Manzagol, P.A.: Extracting and composing robust features with denoising autoencoders. In: Proceedings of the 25th International Conference on Machine Learning (ICML 2008), pp. 1096–1103. ACM, Helsinki (2008). https://doi.org/10.1145/1390156.1390294
25. Wittenberg, D., Rothlauf, F., Schweim, D.: DAE-GP: denoising autoencoder LSTM networks as probabilistic models in estimation of distribution genetic programming. In: Proceedings of the 2020 Genetic and Evolutionary Computation Conference, pp. 1037–1045. GECCO 2020, ACM, New York (2020). https://doi.org/10.1145/3377930.3390180
26. Wong, P.K., Lo, L.Y., Wong, M.L., Leung, K.S.: Grammar-based genetic programming with Bayesian network. In: IEEE Congress on Evolutionary Computation (CEC'14), pp. 739–746. IEEE, Beijing (2014)
27. Wong, P.K., Lo, L.Y., Wong, M.L., Leung, K.S.: Grammar-based genetic programming with dependence learning and Bayesian network classifier. In: Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2014), pp. 959–966. ACM, Vancouver (2014). https://doi.org/10.1145/2576768.2598256
28. Yanai, K., Iba, H.: Estimation of distribution programming based on Bayesian network. In: IEEE Congress on Evolutionary Computation (CEC 2003), pp. 1618–1625. IEEE, Canberra (2003). https://doi.org/10.1109/CEC.2003.1299866