An Estimation of Distribution-like Algorithm based on a Denoising Autoencoder

Alex, Sid, and Chrisantha

School of Electronic Engineering and Computer Science
Queen Mary, University of London
{a.churchill,ssgd,c.t.fernando}@qmul.ac.uk

Abstract. In this paper we present a novel neural-based optimisation algorithm. The algorithm follows the traditional generate-update methodology of Estimation of Distribution algorithms, using a denoising autoencoder to learn the structure of promising solutions within its hidden layer, with the output neurons defining a probability distribution that is sampled from to produce new solutions. The algorithm is shown to outperform a canonical Genetic Algorithm on several combinatorial problems, including the multidimensional 0/1 knapsack problem, MAXSAT and the Hierarchical If and Only If. Analysis shows that the neural network is able to learn interesting structural features of the search space, while the sampling method employed supports continued exploration, enabling optimal solutions to be found on NP-hard problems.

1 Introduction

Estimation of Distribution Algorithms (EDAs) are a growing area in Evolutionary Computation, which attempt to statistically model sections of a search space in order to uncover underlying structure and guide search towards optimal solutions in an efficient manner [1]. At the heart of any EDA lies a model-building algorithm. Examples include Bayesian Networks [2], Markov Networks [?] and K-Means clustering [?]. In this paper we introduce a novel neural-based method for modelling, a denoising autoencoder. An autoencoder is a feed forward neural network, consisting of at least one hidden layer, which is trained to reproduce its inputs from its outputs. Over the course of training the hidden layer learns a compressed representation of the data, which has been used for reconstructing missing data [?] and dimensionality reduction [?] in Machine Learning applications. The algorithm introduced in this paper trains a single autoencoder with promising solutions from a population, learning structural features of the search space in the hidden layer representation. This differs from traditional EDAs such as PBIL [?] or ECGA [?] as an explicit statistical model is not produced. However, the learnt structure can be leveraged to produce new solutions by inputting an existing or randomly generated solution into the network and sampling from the output neurons using a Bernoulli distribution. Results presented in Section 5 show that the autoencoder method is able to outperform a canonical Genetic Algorithm (GA) across a range of combinatorial and hierarchical problems.

2 Background

EDAs (also known as Probabilistic-Model-Building Genetic Algorithms) are population based optimisers, that typically replace genetic operators with a statistical model. The rationale behind the model building and related linkage learning approach is that dependencies between variables can be captured and preserved, which can be easily lost in standard Evolutionary Search, and new solutions generated around promising structures. Early work on EDAs concentrated on methods that explicitly modelled the probabilities of features occurring independently in a population of genotypes. These include the compact Genetic Algorithm [**?**], PBIL [**?**] and Univariate Marginal Probability methods[**?**]. Improved success was found by modelling multivariate dependencies using clustering algorithms (ECGA) [**?**], Bayesian Networks [2], Markov Networks [**?**] and tree structures [**?**], among others.

The use of the autoencoder model in this paper is motivated by its potential to learn high-dimensional non-linear dependencies in data, while maintaining a low computational cost in terms of training time. Recently there has been interest in neural-based methods in an EDA context for multi-objective optimisation. A Growing Neural Gas (GNG) was used as a model in [**?**], employing a competitive Hebbian learning rule to cluster data without having to pre specify the number of groups. A shallow Restricted Boltzmann Machine (RBM) was used to model high dimensional data in [**?**], beating the state-of-the-art on several multi-objective continuous benchmark problems. An autoencoder is another neural-based method for the unsupervised learning of features and has hitherto not been applied to combinatorial or continuous optimisation.

A second motivation for this approach is to investigate methods in which Evolutionary Algorithms can be implemented using neural structures. The *Neural Replicator Hypothesis* [**?**] proposes that evolutionary processes could operate in the brain at ontogenetic timescales. In [3], a (1+1)-ES is implemented in a network of spiking neurons. Adding Hebbian learning enabled linkages to be found between features, and the 128-bit HIFF problem to be solved.

3 Methods

3.1 Autoencoders

A standard autoencoder consists of an encoder and a decoder. The encoder performs an affine transformation followed by an element-wise non-linear operation. The mapping performed by the encoder is deterministic and can be described as:

$$h_\theta(\mathbf{x}) = f(\mathbf{W}\mathbf{x} + \mathbf{b})$$

The non-linear activation function f can be any one of the standard non-linearities used in artificial neural networks (sigmoid,tanh). The encoder parameters are $\theta = \{\mathbf{W}, \mathbf{b}\}$.

The decoder is also a deterministic function and operates on the encoded input $h_\theta(x)$ to produce the reconstructed input:

$$r_{\theta'}(\mathbf{h}) = g(\mathbf{W}'\mathbf{h} + \mathbf{b}')$$

The decoder parameters are $\theta' = \{\mathbf{W}', \mathbf{b}'\}$. The output non-linearity, h can vary depending on the whether the input data is binary or continuous. The outputs of the decoder are generally not considered as exact reconstructions, but parameters for some distribution $p(X|Z = \mathbf{z})$.

3.2 Training

For combinatorial problems, the inputs $\mathbf{x} \in \{0,1\}^d$. The decoder must produce a binary output, therefore reconstructed outputs z are considered to be parameters for the distribution $X|\mathbf{z} \sim \mathcal{B}(\mathbf{z})$ where $\mathcal{B}$ is the bernouilli distribution. The autoencoder is trained by minimizing the loss function w.r.t to the model parameters using stochastic gradient descent:

$$\theta^*, \theta'^* = \arg\min_{\theta,\theta'} \frac{1}{n}\sum_{i=1}^{n} \mathcal{L}(\mathbf{x^{(i)}}, \mathbf{z^{(i)}})$$

For binary inputs, we use the cross-entropy loss function. The cross-entropy loss function can be viewed as the negative log-likelihood of the training vector $\mathbf{x}$ given the parameters for the bernouilli distribution $\mathbf{z}$. The loss function $\mathcal{L}$ is defined as:

$$\mathcal{L}(\mathbf{x}, \mathbf{z}) = -\sum_{\mathbf{j}}[\mathbf{x_j}\log\mathbf{z_j} + (\mathbf{1} - \mathbf{x_j})\log(\mathbf{1} - \mathbf{z_j})]$$

Empirical evidence suggests that training autoencoders by only minimizing reconstruction loss might not be enough to learn useful representations in the hidden layer. An autoencoder with enough capacity can easily learn the identity function, thus reconstructing the input perfectly each time, which is not very useful. In order to force the network to learn interesting transformations, an information bottleneck must be imposed during training. A common approach is to set the number of hidden units to be smaller than the dimensionality of the data. This ensures that the network must discover structure in the inputs in order to reconstruct with high accuracy. Tying the weights of the encoder and decoder networks to be the same is another method used to guide training. Apart from these techniques, the standard regularisation techniques like L1 and L2 weight [?] decay are also employed while training autoencoders.

3.3 Denoising Autoencoders

With the revival of interest in neural nets and deep neural architectures, several new variants of autoencoders have been proposed which differ in the kind of regularisation used during training [?,?]. The denoising autoencoder is a particularly interesting variant of the classical autoencoder that tries to reconstruct a *clean* version of a noisy example, thus *denoising* noisy inputs.

The input $\mathbf{x}$ is first corrupted using a stochastic corruption process $\tilde{\mathbf{x}} = \mathbf{q}(\tilde{\mathbf{x}}|\mathbf{x})$. The corrupted input is then mapped to a hidden representation $\mathbf{h}$ by the encoder. Finally, the decoder tries to reconstruct the uncorrupted input $\mathbf{x}$ from $\mathbf{h}$. The denoising autoencoder is trained by minimizing the cross-entropy loss on the training set using stochastic gradient descent. The functioning of the denoising autoencoder can be interpreted in two ways. One view is that forcing the network to reconstruct clean inputs from corrupted inputs acts as a regulariser. It ensures that training must converge to networks that learn transformations more complex than the identity function. Denoising autoencoders also have a geometric interpretation [**?**]. A denoising autoencoder learns the distribution $p(X|\tilde{X})$. This distribution maps noisy inputs to clean examples seen in the training set. Therefore the denoising encoder maps points from regions of low probability to regions of higher probability. These points with higher probability mass are said to form a low dimensional manifold according to the manifold assumption [**?**]. Under this interpretation, denoising autoencoders can be used for learning manifolds in input space.

3.4 Autoencoders and Genetic Algorithms

EDAs try to improve GA convergence by building a distribution of the most promising solutions at each generation and sampling a new population of solutions from this distribution. Therefore the two main requirements for building an effective EDA are efficient distribution estimation from training examples and efficient sampling from the learned distribution[citation]. In this work we propose an alternative approach to model building and sampling using denoising autoencoders.

Using the dAE to model conditional distributions has the advantage that the network is trained with standard backpropagation, which is deterministic and does not involved any sampling. This leads to quick and efficient training of models at each generation. dAEs are also advantageous because the output distribution of the network given a reconstruction $\mathbf{z}$ is $X|\mathbf{z} \sim \mathcal{B}(\mathbf{z})$. Therefore sampling new solutions involves sampling from a bernoulli distribution which is much simpler compared to sampling from more complex distributions [examples,citations].

The proposed algorithm differs from a standard EDA in the fact that we model conditional distributions $p(X|\tilde{X})$ instead of distributions of the input $p(X)$. Although denoising was introduced as a regulariser for learning better networks [**?**], we find it particularly well suited for modelling in a GA. It forces bad samples towards better samples from the distribution, by means of the conditional probability distribution which maps inputs that are far from the 'good' solutions towards them.

Another challenging problem associated with EDAs is the problem of maintaining a good variance in the newly sampled population, so that the GA can explore the space effectively. The problem of EDAs converging to a few points has been observed in several cases [citations]. With the conditional distribution, we can control the variance in the output, by choosing inputs that are reasonably diverse. This ensures that the new population has enough variance to perform search effectively.

3.5 Optimisation Algorithm

The pipeline of the *Denoising Autoencoder Genetic Algorithm* (DAGA) is similar to other EDAs and is inspired by methods in HBOA [2]. A population of solutions is maintained, P_t, and updated at each iteration, t. An initial population of solutions, P_0, is drawn from a uniform distribution. These solutions are evaluated and the fittest unique x% are selected (i.e. truncation selection) to be in the training set, $\hat{P}_t$. The Denoising Autoencoder, D_t, is then trained with $\hat{P}_t$ for e epochs. Following training, a new set of solutions, S_t, is selected from P using tournament selection (with replacement). Each member of S_t is inputted to D_t, and the output vector, y, is sampled from using a binomial distribution, to produce a new solution. This solution is included in P_{t+1} if it is better than its closest neighbour according to Restricted Tournament Selection (RTR) [2]. Pseudocode for DAGA is presented in Algorithm 1.

4 Experiments

Experiments intro blurb.

4.1 Multi-dimensional Knapsack Problem

Here we wish to choose of subset of N items to maximise the total value of items, z, while satisfying m constraints. We wish to maximise:

$z = \sum_{j=1}^{N} v_j x_j$, subject to $\sum_{j=1}^{N} w_{ij} x_i \leq c_i, i = 1, ..., m$

$x_i \in \{0, 1\}, j = 1, ..., N$.

Two different instances are used in the results presented below. The first is the Weing8 instance [?], which has 105 items and two constraints (optimal solution is 602,319), and the second is a randomly generated instance with 500 items and one constraint (optimal solution is 100,104). Both instances are available in the supplementary material. If any of the m constraints are violated, the total value of chosen items is multiplied by -1.

4.2 Hierarchical If and only If

The Hierarchical If and only If (HIFF) problem was created by Watson et al., as an example of a function which is not separable [?]. It is a pathological function for a hill climber because of a very large number of local optima. The HIFF is structured as a balanced binary tree [2], where at each lower level a transfer function is applied to consecutive non-overlapping 2-bit partitions, $00 \rightarrow 0$, $11 \rightarrow 1$, anything else $\rightarrow$ null, to define the string in the layer above. At each level, 00 and 11 blocks contribute to the fitness by 2^{level}. Two global optima exist, a string of all 1s and all 0s. In the results section, DAGA is applied to the 128-bit HIFF. In order to remove any possible bias towards an all 1s solution, solutions from DAGA have a random bit mask (fixed before each trial) applied to them before evaluating fitness.

4.3 Royal Road

The Royal Road function as defined by Mitchell et al. [?], divides a bit string, x, into a number of equally sized, non-overlapping partitions, $z_i \in [x_i, ..., x_{i+n}])$. If all of the bits in the partition match a predefined pattern, s_i , then the partition's fitness contribution is added to the overall fitness. The existence of these "Building blocks" was meant to act as a "royal road" for GAs compared to hill-climbers but bad alleles hitchhiking to good partial solutions slows down convergence speed. The 128-bit problem with 8-bit partitions is defined as,

$$f(x) = \sum_{i=1}^{16} \delta_i(x) o(s_i), \text{where } \delta_i(x) \begin{cases} 1, & \text{if } x \in s_i \\ 0, & \text{otherwise} \end{cases}$$

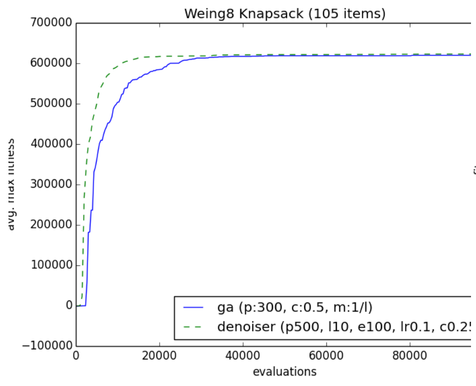
If a string contains two instances of s, a fitness of 16 is achieved. If all instances are found, fitness = 128. s_i instances are blocks of all 1s but as with the HIFF, a random mask is applied to the output of DAGA before fitness evaluation.

4.4 MAXSAT

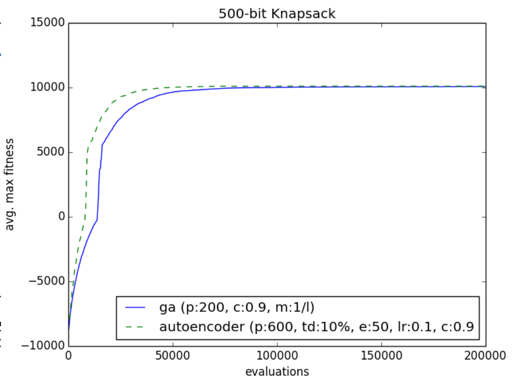
5 Results

DAGA is compared to a canonical generational Genetic Algorithm (GA) on the problems described in Section 4. The GA employs tournament selection to select parents, two point crossover for recombination (with probability p_c) and a probability, p_m, of a bit flip mutation at each allele. The GA is implemented in the DEAP framework [?], while DAGA is implemented in Python, using Theano for the *dA*. Programming code is available in the online supplementary material. For each experiment a large parameter sweep was performed on both DAGA and the GA and the best found configurations were chosen for comparison. Figure 1 presents graphs showing the mean fitness (averaged over ten trials) of the best solution in the population at each iteration for the two algorithms on the 5 problems. Details of the best solutions found are given in Table **??**.

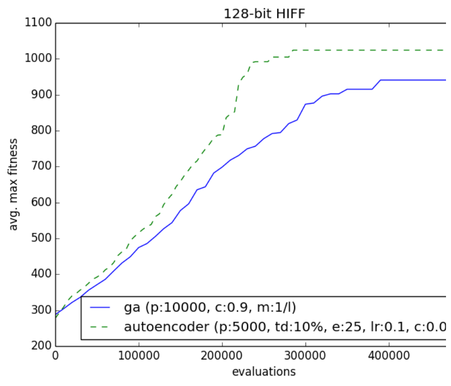
From Figure 1 we can immediately see that DAGA finds significantly better solutions or optimal solutions significantly faster than the GA. On three of the five experiments (both knapsacks and MAXSAT), DAGA finds significantly better solutions than the GA, within the given evaluation limits. On the Weing8 knapsack instance DAGA reaches the optimum solution 8 out of 10 attempts, and on the 500-item instance 7 out of 10 attempts. On the MAXSAT instance, DAGA also reaches the optimum 70% of the time. On all three of these problems the GA is unable to locate the optimal solution a single time. On the HIFF,



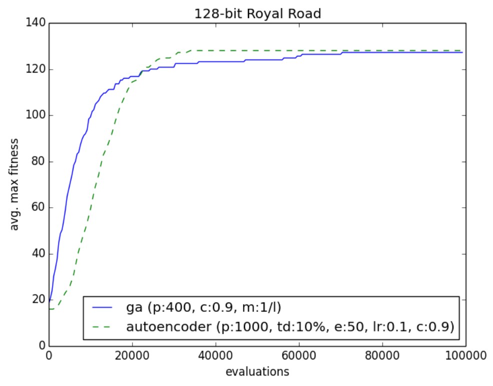
(a)



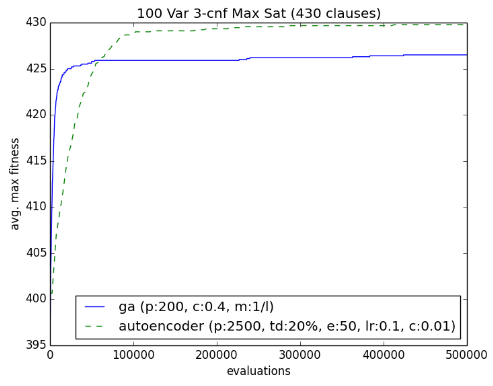
(b)



(c)



(d)



(e)

Fig. 1: Plots showing for each experiment the mean fitness of the best solution in the population over time, averaged over 10 trials.

Experiment	Algorithm	Min	Max	Mean 1Q	Mean 2Q	Mean 3Q	Mean 4Q	Success %
MAXSAT	GA	424.0	429.0	425.9	426.2	426.3	426.5	0%
	AE	429.0	430.0	429.1*	429.56*	429.67*	429.78*	70%
HIFF	GA	832.0	1024.0	505.2	756.6	915.2	940.8	50%
	AE	1024.0	1024.0	568.4*	992.0*	1024.0*	1024.0	100%
Knapsack Weing8	GA	619568.0	621086.0	609728.9	619170.1	619856.0	620099.2	0%
	AE	621086.0	624319.0	617593.2*	621446.8*	622056.4*	623124.3*	80%
Knapsack 500	GA	10047.0	10093.0	9637.7	10011.8	10055.8	10074.2	0%
	AE	10096.0	10104.0	10020.0*	10101.4*	10102.7*	10102.7*	70%
RR	GA	120.0	128.0	120.0	124.0	127.2	127.2	90%
	AE	128.0	128.0	122.4	128.0	128.0	128.0	100%

Table 1: Results for DAGA and the GA on the 5 problems. Showing the value of the minimum and maximum solution returned at the end of search, and the mean best solution in the population after 25%, 50%, 75% and 100% of evaluations had been completed, across 10 trials. Stars indicate that the mean result for DAGA is significantly different from the GA according to a Wilcoxan Rank Sum test ($p < 0.05$).

DAGA locates the optimal solution every time, while the GA locates it on only half of the trials. On the Royal Road, DAGA again locates the global optima on every trial, while the GA finds it on 9 out of 10. These results show that on a wide range of problems DAGA is able to find higher quality solutions and more consistently locate the optimum compared to a GA.

As well as frequently obtaining better quality solutions at the end of the optimisation process, DAGA also finds significantly better solutions than the GA at earlier stages in the optimisation process on a number of the tested problems. Table **??** shows the mean best solution found after different numbers of function evaluations have passed. From this we see that on all of the problems, apart from RR, DAGA finds significantly better solutions than the GA after a quarter of function evaluations have passed. On the HIFF and RR functions, the GA is able to locate optimal solutions eventually but not consistently. DAGA is statistically significantly better after 25%, 50% and 75% of evaluations have passed on the HIFF. On this difficult problem, DAGA finds the optimum after 230,000 evaluations on average, with 200,000 in the fastest case and 285,000 in the slowest. At its fastest, the GA finds the optimum after 300,000 evaluations (mean 332,000 on the successful attempts), although on half the trials it fails to reach the optimum after 500,000. On RR, DAGA has a better mean score from the first quarter onwards, but the GA regularly reaches the optimum solution and their scores are not significantly different statistically (p=0.13 from 25% to 50%).

Table **??** presents the parameter values for the chosen configuration of DAGA and the GA used to solve each problem. A noticeable trend is that two of the

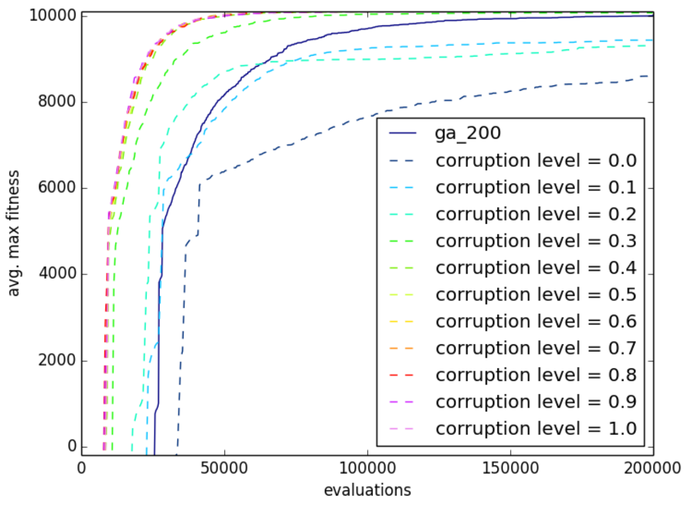


Fig. 2: Showing the fitness of the best solution in the population over time of DAGA with different corruption levels on the Knapsack 500 problem, averaged over 5 trials.

problems use a very high corruption level (Knapsack 500 and RR) while the others use a low number. The high corruption level (CL) instances could be seen as easier to solve, as the knapsack instance, although larger than Weing8, has only one set of constraints and RR has few local optima. Figure 2 shows the effect of varying CL from 0.0 - 1.0 on the 500-bit knapsack. It is clear that CL has a large impact on the convergence speed, with high CLs finding high quality solutions very quickly.

6 Discussion

7 Conclusion

Acknowledgments. The work is funded by the FQEB Templeton grant "Bayes and Darwin", and the FP-7 FET OPEN Grant INSIGHT.

References

1. Pelikan, M., Sastry, K., Cantú-Paz, E.: Scalable optimization via probabilistic modeling. Springer (2006)
2. Pelikan, M.: Hierarchical Bayesian optimization algorithm. Springer (2005)
3. Fernando, C., Goldstein, R., Szathmáry, E.: The neuronal replicator hypothesis. Neural Computation **22**(11) (2010) 2809–2857