Automated development of latent representations for optimization of sequences using autoencoders *

Piotr Kaszuba Maciej Komosinski Agnieszka Mensfelt

Poznan University of Technology
Institute of Computing Science
Poznan, Poland
maciej.komosinski@cs.put.poznan.pl

Abstract

In this paper, we propose an automated method for the development of new representations of sequences. For this purpose, we introduce a two-way mapping from variable length sequence representations to a latent representation modelled as the bottleneck of an LSTM (long short-term memory) autoencoder. Desirable properties of such mappings include smooth fitness landscapes for optimization problems and better evolvability. This work explores the capabilities of such latent encodings in the context of optimization of 3D structures. Various improvements are adopted that include manipulating the autoencoder architecture and its training procedure. The results of evolutionary algorithms that use different variants of automatically developed encodings are compared.

Keywords: genetic representation, autoencoder, latent representation, fitness landscape, evolutionary design

1 Introduction

This work introduces a method for automated development of representations of sequences and investigate it in the context of the challenging problem of evolutionary design. Evolutionary design is the optimization of structures, often three-dimensional – which can be either active (equipped with a control system) or passive – with the use of evolutionary algorithms. Its applications include optimization of e.g., circuits, mechanical parts, neural networks, or even the creation of art [1]. Evolutionary design supported by simulation environments allows to automate and substantially reduce costs of developing solutions to real-world problems. Depending on the application, it can provide better results than a human expert would. Nevertheless, the optimization of 3D structures can be considered one of the hardest optimization problems due to the discrete-continuous nature of the problem, costly multi-criteria evaluation of the solutions, and the infinite size of the solution space. Furthermore, in contrast to many typical combinatorial optimization problems, the genotype-to-phenotype mapping is often very complex, which leads to a rugged fitness landscape.

Despite the crucial importance of the genetic representation for the success of evolutionary search and the multiplicity of the existing representations for active and passive 3D structures [6,7,11,20,21], there are no general guidelines for developing efficient representations. The currently dominant approach of handcrafting the genetic representations is rather an art than a science; the researchers

*The final version of this paper appeared in 2021 IEEE Congress on Evolutionary Computation (CEC), 2021. http://dx.doi.org/10.1109/CEC45853.2021.9504910.
The third author was supported by the Faculty of Computing, Poznan University of Technology, through the funds provided by the Ministry of Science and Higher Education.

solving this difficult task have to rely on their intuition, possibly introducing human bias and developing inefficient representations. In this paper, we introduce an alternative for this method, which addresses all of the aforementioned disadvantages and has the potential to significantly increase the effectiveness of the evolutionary design. Our approach is based on the use of an autoencoder to automatically learn the genetic representation with desired properties. An autoencoder is an unsupervised learning technique based on artificial neural networks used to discover the efficient representation of the data. Autoencoders were first introduced in 1986 [18] and have been widely adapted and extended since then. Their applications include dimensionality reduction, feature learning, data denoising, image generation, modelling products and clients in recommendation systems, anomaly detection [19], transfer learning, and representation learning for evolutionary algorithms [16]. Autoencoders usually have a few feed-forward fully connected layers and are trained to reconstruct their input at the output. They are designed to learn complex, nonlinear relationships in data.

To obtain a learned genetic representation, in this work an autoencoder is trained to reconstruct genotypes in some existing ("native") encoding. The optimization search can then be performed in the latent space of the autoencoder bottleneck using the decoder to map back from the latent space to the native genotype space. The use of an autoencoder to develop a genetic representation can not only free the researchers from this daunting task, but also provide genetic representations possessing the desirable properties of high locality and evolvability. The high locality is obtained when small changes in the genotype result in small changes in the phenotype [17]. Without this property, the fitness landscape may be rugged and in consequence, the evolutionary search may become similar to the random search. If the genetic representation ensures the topology of the solution space in which similar solutions are close to each other, then the fitness landscape possesses the property of global convexity, which facilitates the evolutionary search. This property can be assessed using fitness-distance correlation (FDC) analysis [8]. High values of FDC indicate that the shape of the fitness landscape may be convex. The other desired property that can be provided by the genetic representation is evolvability – the ability to produce improvement when random variations occur. Higher evolvability can be achieved when individuals with higher fitness are of higher probability in the genotype space, the ruggedness of the fitness landscape is low, or when better solutions are always in proximity – e.g., within the reach of few mutations. High evolvability corresponds to the high probability of producing an improvement through genetic variation. The more often an improvement occurs, the quicker the evolutionary algorithm should progress and the higher the achieved fitness can be.

2 Methods

2.1 Simulation environment

The experiments are performed using the Framsticks simulation software [12,13] which is available as a native binary (compiled from C++) for all major desktop and mobile operating systems. We use the python interface [15] to perform evolutionary optimization with DEAP [2]. Framsticks simulates creatures (robots, agents) with bodies (physical constructs) and, optionally, brains (neural networks of arbitrary topology and various types of neurons). Two kinds of body models are supported: one consisting of "sticks" (elastic rods), and one consisting of rigid, solid shapes (cuboids, ellipsoids, cylinders). In this work we use the former body model. There are several genetic encodings available in Framsticks – ranging from simple, direct low-level representations to sophisticated, indirect high-level ones [14]. Each encoding has its own genetic operators (mutation, crossover, repair). In the experiments reported here we use two very different native genetic representations:

- *f9* is a very simple encoding where genotypes consist of only six letters, each letter corresponding to one direction of movement in the 3D space. The phenotype (body) is constructed as if it was drawn in "turtle graphics" by executing successive letters in the genotype in the absolute coordinate system: `L` for left, `R` for right, `D` for down, `U` for up, `B` for back, and `F`

Figure 1: Two examples of simple genotypes in the native *f9* representation: `RLURF` and `DLLLLLDBUFBLU`. Left pair: phenotypes before simulation, right pair: during simulation.

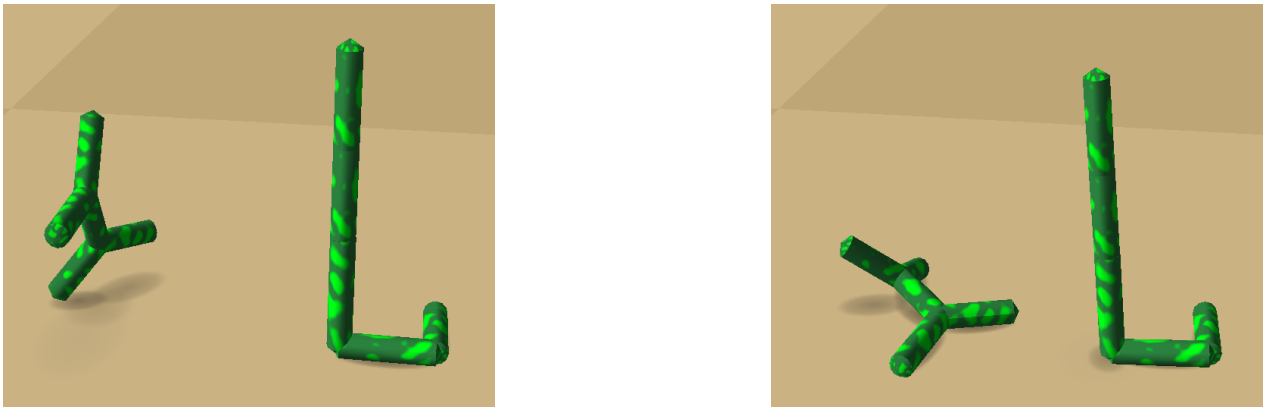

Figure 2: Two examples of simple genotypes in the native *f1* genetic representation: `LLRRRRRRX(X,RX(X,X))` and `Rff(,LrffX,LLRfXLLLRR(,,LXrrrXLX))`. Left pair: phenotypes before simulation, right pair: during simulation.

for forth, as illustrated in Fig. 1. The length of sticks is constant; neural networks cannot be encoded in this representation. Any sequence built from the six letters is a valid genotype. Mutation changes one randomly selected symbol into another, random one, or deletes one random symbol, or adds one random symbol.

- *f1* is more sophisticated, but contrary to *f9*, due to the recursive nature of this representation, encoded bodies are restricted to tree structures. The most important letters in the genotype are `X` to create a stick, parentheses `( )` to indicate a branching, a comma `,` to separate individual branches, and additional letters that modify specific properties of sticks (like characters `L` and `l` to lengthen or shorten sticks, `R` and `r` to rotate the branching plane by ±45 degrees, or `F` and `f` to adjust the friction coefficient). Neural networks of any topology can be encoded as well and connected to the body by sensors and effectors, but such neural control systems are not used in the experiments reported here – genetic operators are not allowed to include neurons in genotypes. Some sequences of letters in this encoding do not constitute valid genotypes and cannot be translated into phenotypes without applying an additional repair procedure. However, mutation and crossover operators try to always produce valid genotypes by observing constraints of the representation (e.g., by matching parentheses properly). Mutation modifies random aspects of the genotype (adds or removes parentheses, commas, or allowed letters). Sample agents are shown in Fig. 2.

In both *f9* and *f1*, two-point crossover is used which swaps randomly selected substrings in parents. The genotype-to-phenotype mappings of both genetic encodings differ significantly. In *f9*, the influence of individual genes depends less on their context than in *f1*. However, even in *f9* the influence of individual genes on the phenotype and its fitness is far from obvious due to the elasticity of body parts and their mutual dynamical interaction in simulation.

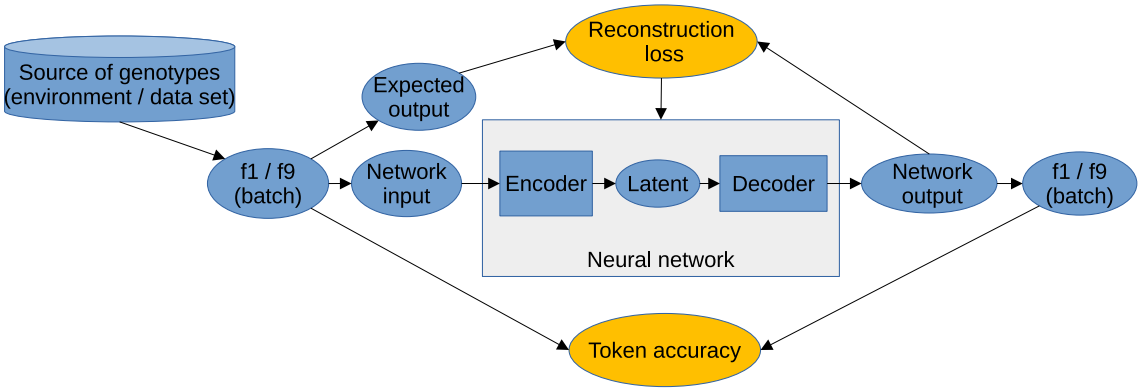

Figure 3: Basic architecture of the system. Blue shapes depict the genotype processing pipeline while yellow ones refer to quality measures. Processed objects are ovals, and autoencoder components are rectangles.

2.2 General architecture

Fig. 3 shows the general architecture of the employed system. New genotypes are generated either by randomly assembling a sequence of symbols or by mutating several times randomly picked genotypes. The length distribution of genotypes is controlled by defining probabilities over lengths from 1 to a specified limit. Since the number of possible sequences grows exponentially with their length, experiments also use exponential distributions of genotypes to reflect the distribution of considered genotype space. Data converters are mandatory parts of the pipeline that transform the data by adding start ('S') and stop ('T') tokens to genotype strings, preparing the autoencoder input data and expected output data, and converting the autoencoder output back to strings.

The neural network is the long short-term memory (LSTM) [5] autoencoder. The autoencoder input data is a sequence of integers corresponding to symbols of the native encoding (*f1* or *f9*) interpreted by the autoencoder's embedding layer as categorical inputs. Expected output data is a sequence of one-hot encoded characters, as neural network output has the same structure but contains probability distribution over characters for each sequence element. Genotype strings are obtained from the autoencoder output by picking the most probable character for each sequence element up to the maximum length of a sequence or until the stop token is picked. If the autoencoder generates multiple start tokens, the first sequence between the start and stop tokens is considered the output. The autoencoder consists of two main parts: the encoder and the decoder, as shown in Fig. 4. In preliminary experiments, autoencoders with 1 and 2 LSTM layers were tested. The 2-layer architecture provided higher reconstruction accuracy, so it is used.

The basic encoder flow is as follows:

1. Inputting sequence consisting of integers.
2. Embedding the sequence into k-dimensional space. Its purpose is to find a space with values more suitable for LSTM to process.
3. Processing the sequence with LSTM that contains C_E cells and fetching the final hidden state (i.e., the recursive signal) and the cell state. These states summarize the sequence.
4. Batch normalization of a vector obtained by concatenation of the LSTM final states. This serves as encoder regularization.
5. Executing in parallel:
   - Feed-forward fully connected layer with C_D neurons and hyperbolic tangent activation returning h.

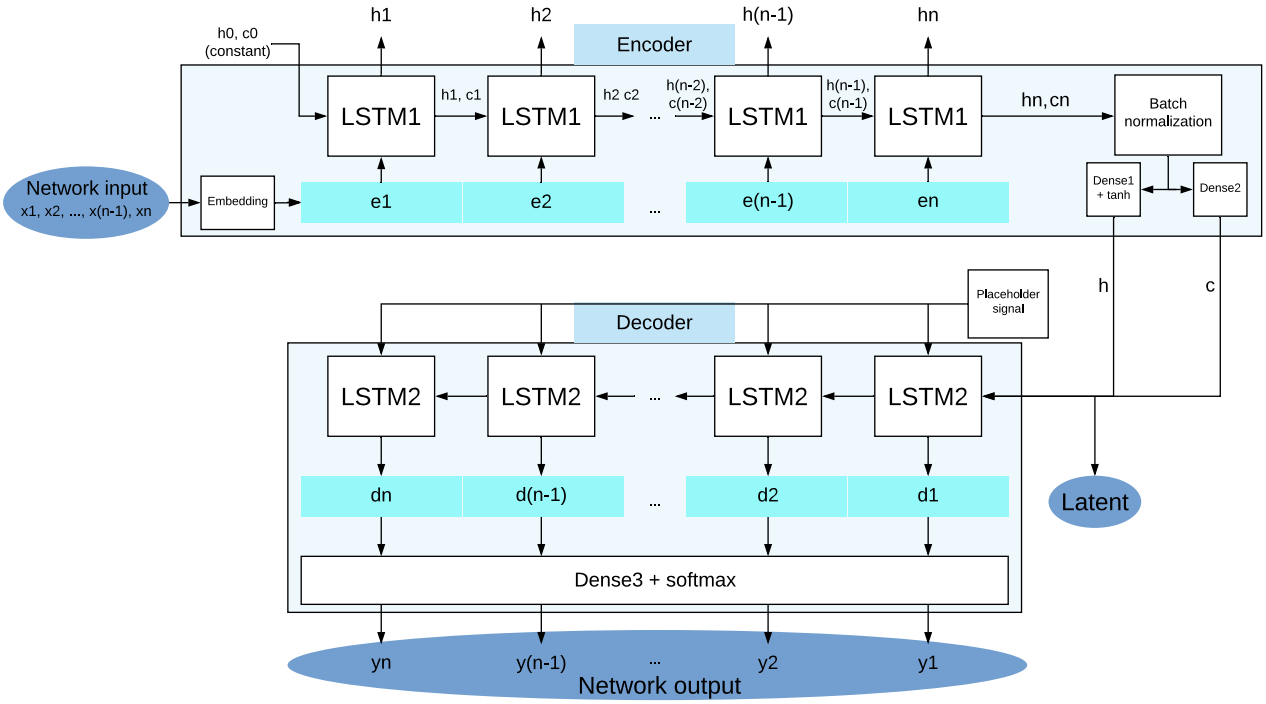

Figure 4: Architecture of one-layer, unidirectional autoencoder. Rectangles and arrows depict internal autoencoder components and processed signals. Blue ovals correspond to input/output signals.

- Feed-forward fully connected layer with C_D neurons and linear activation returning c.

LSTM states are first concatenated and then obtained from fully connected layers in order to make the dimensionality of the latent representation independent from both the number of state pairs concatenated (for bidirectional LSTM) and the cells in the encoder LSTM.

6. Encoder output is h, c which constitutes a $2C_D$-dimensional latent space.

The concatenated encoder output is the latent representation of the input sequence. Decoder signal flow is the following:

1. Inputting h, c.
2. Processing the *placeholder signal* with LSTM that contains C_D cells initialized by h, c and collecting sequence of outputs. The output sequence has information about the order of symbols already extracted from h, c – the following steps described below may transform vectors encoding individual sequence symbols independently and in parallel.
3. Every step in the collected sequence is passed through the same feed-forward fully connected layer with a softmax activation. It transforms the LSTM output space and reduces the dimensionality down to the number of available symbols (also the number of neurons in this layer). Softmax transforms layer activation into a probability distribution over symbols proportional to the exponentials of their activations.
4. Decoder output is the sequence processed by the last layer.

2.3 Autoencoder training

The autoencoder is trained until it achieves the token accuracy of 0.99. Each epoch consists of 100 steps with the batch size equal to 50 and uses the sample of 5000 genotypes. The autoencoder

is optimized with Adam optimizer [9] (default parameters) and the learning rate of 0.001. Initial weights are drawn from the He Normal distribution [4]. The primary loss function which serves as an assessment of reconstruction quality is categorical crossentropy. Experiments conducted in this work showed that it is worth using the L1 and L2 regularization losses on the activity of the latent space. L1 introduces sparsity pressure to the autoencoder – unused dimensions are pushed toward zeros. L2 keeps activations relatively close to the origin – genotypes are packed closer together. The regularization weight of both terms is set to $2 \cdot 10^{-6}$.

To obtain a latent genetic encoding ensuring a smooth fitness landscape, we pass the information about the phenotype space to gradients that update the neural network. For this purpose, we use the similarity measure implemented in Framsticks [10], which provides information about structural and geometric distances between phenotypes. The encoder inputs are discrete sequences, and the vector of distances between every pair of sequences can be obtained from the dissimilarity measure. This vector is used as a constant term in the loss function. The encoder output is passed through a node that calculates a vector of Euclidean distances between every pair of returned points in the latent space. The locality loss term is based on the Pearson correlation coefficient *cor* of both vectors and is defined as $w \cdot (1 - cor)$, where w is the weight of the locality term. Gradients of the locality term w.r.t. the encoder parameters are based on the correlation of distances in the latent space and the phenotype space, assuming that the dissimilarity measure provides accurate phenotype distances. Applying these gradients makes the sequence-to-latent mapping generate a topology in the latent space that is more similar to the topology of the phenotype space. Vectors of distances are obtained for all genotypes in a batch. This approach is analogous to a semantic loss for program embedding in the latent space [16]. Another possibility is to use the difference of fitness values instead of the phenetic dissimilarity measure; organizing the space of solutions so that solutions with similar fitness values are neighbors is the most desirable for optimization. However, this approach may be less general than employing the phenetic dissimilarity measure, because the learned latent representation may be too adapted to a particular kind of fitness functions, and may turn out to be inefficient in the optimization of other fitness functions. In the experiments, loss based on the dissimilarity measure is called *locality phenotype* and loss based on the fitness function is called *locality fitness*. Following the preliminary experiments, the weight of the locality term was set to 3. The scheme of locality loss application is shown in Fig. 5.

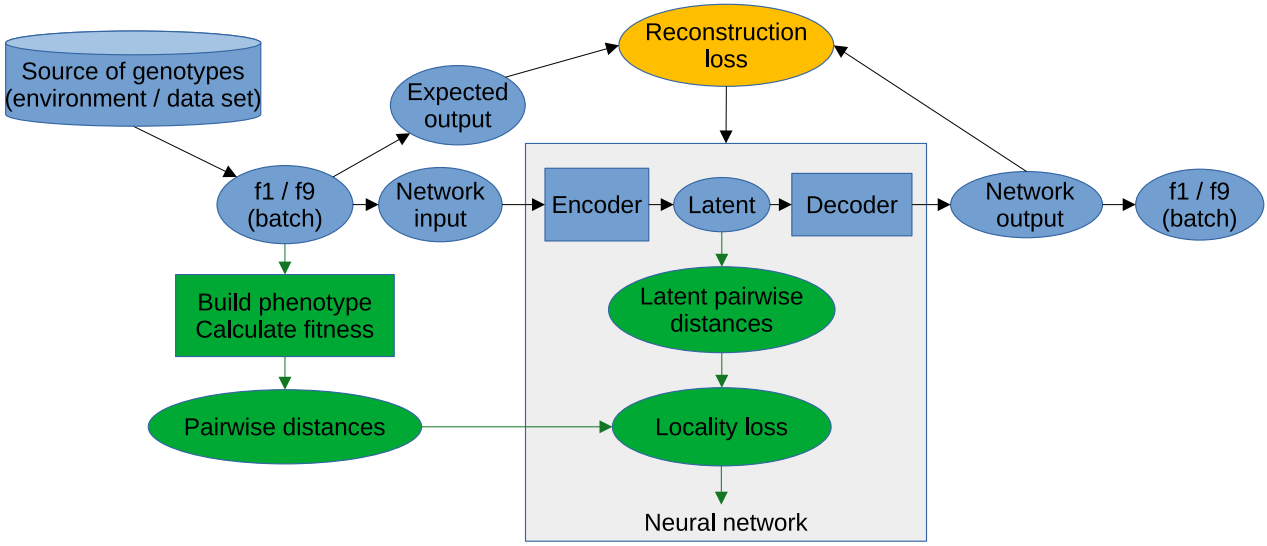

Figure 5: Application of locality loss in the system. Blue and yellow shapes depict the basic sequence processing pipeline and calculated measures. Green shapes are additional components that extend the basic architecture. Processed objects are ovals, and autoencoder and system components are rectangles.

For the purpose of sampling points in the latent space and decoding them to the native sequence representation, a latent distribution is learned. The latent distribution is modeled as a multivariate normal distribution. It is centered at the origin for the mutation operator and at means of dimensions for the independent sampling of individuals from the latent space. Means and the covariance matrix are learned on 5000 sampled genotypes in the native encoding that were passed through the encoder. This covariance matrix is later used in the mutation operator.

A sequence-latent point (SLP) *slp* for a point p in the latent space is defined as $slp = enc(dec(p))$, where *enc* is the encoder with the sequence-to-network-input converter, and *dec* is the decoder with the network-output-to-sequence converter. If the autoencoder is able to reconstruct every sequence correctly, every SLP is also a fixed point of the function $f(x) = enc(dec(x))$:

$$slp = f(slp)$$

Sequence-latent points are generated by the encoder. The decoder predictions around SLPs should be more reliable, since the decoder learned to predict genotypes based on encoder outputs. In turn, the latent space in an SLP proximity may have better properties such as a less rugged local landscape and a higher density of genotypes that are valid and similar to each other. Since the SLPs are the product of the sequence-to-latent mapping (the encoder) that is directly trained to organize the space of solutions, these SLPs are expected to follow the global topology of solutions that is induced by *locality loss* more reliably than points that are not SLPs. Ideally, for any p, its *slp* would be as close as possible in the latent space. That would mean that the decoder indirectly (through the encoder) learned to organize the space of solutions.

2.4 Evaluation methods

The overall quality of the approach is assessed by measuring evolutionary optimization results and the fitness–distance correlation value in the set of randomly generated solutions. The chosen metric is the fitness of the best found result for every generation across algorithm runs. Evolution in the latent space is based on the latent-to-sequence mapping, i.e., the decoder. The optimization assumes traversing the latent space to obtain a point that maps into the optimal phenotype. For evaluation, a point in the latent space is translated to the native encoding by the decoder and the data converter, and then into the phenotype. Both autoencoder components are available for the latent genetic operators.

The experiments employ a simple evolutionary algorithm with elitism. The population size is set to 50 and the algorithm runs for 250 generations. Initialization of the starting population is done with randomly generated genotypes according to the length distribution. To obtain latent genotypes, genotypes in the native genetic representation are passed through the encoder. After the evaluation of the initial population, for each generation, the algorithm:

- selects a new population using tournament selection of size 3,
- applies the crossover to the genotypes randomly arranged in pairs with 15% probability. The crossover produces 2 offspring that replace their parents,
- for every genotype decides with 60% probability whether it should be mutated,
- preserves elitism: if the best genotype found so far is not present in the population, the best genotype replaces one random genotype in the population,
- evaluates genotypes.

Mutation samples a vector from the latent distribution with the learned covariance matrix scaled by the mutation magnitude parameter (set to 0.2 in the experiments). The sampled vector is added to the individual that is being mutated. Mutated latent vector is then passed through the decoder to obtain genotype in the native encoding. This process is performed 200 times in a single batch.

Duplicate and invalid genotypes are removed. Mutation returns a latent vector that corresponds to a randomly selected genotype from the set of valid ones. If it is not possible, mutation does not change the original latent genotype. Crossover is a linear combination of parents p_1 and p_2, so a child $c = wp_1 + (1 - w)p_2$, where w is a random value from the range of $[0.1, 0.9]$. 400 offspring are collected, decoded to the native representation, and 2 genotypes randomly selected from the set of valid ones are returned. The crossover operator replaces both parents with the new offspring.

The sequence-latent points are considered an important concept when the decoder is learned indirectly through the encoder-based locality loss. An experimental set of genetic operators is designed around the SLPs. The SLP operators follow a similar procedure to the base latent genetic operators. After obtaining the modified point p in the latent space, they transform it into the corresponding sequence-latent point by applying the decoder to p and the encoder to the result. The procedure is done for every sampled point in the set of valid genotypes before picking the mutant. The picked, transformed point is considered a mutated individual. If the autoencoder correctly reconstructs genotypes, the evaluation result after the transform does not change.

To assess whether optimization in the learned representation occurs, results of the evolutionary algorithm are compared to the evolutionary algorithm in which most of the selection pressure is removed from the algorithm – tournament selection size is set to 1, but elitism is still present (Fig. 7 discussed in detail in the following section). Baseline results are also shown for the untrained (random) model.

The relative problem difficulty for the latent representations is compared by inspecting the FDC (Fig. 8, again discussed in more detail in the following section). FDC is calculated as Spearman's rank correlation between fitness scores of solutions and their average distance from not worse solutions in the latent space. Individuals are sampled from the latent representations and evaluated to obtain their fitness. Distances are calculated according to the Euclidean metric.

3 Experiments and Results

The optimization goal for simulated 3D structures is the maximization of the vertical position of their center of mass, as shown in the right panels in Figs. 1 and 2. The maximum length of a genotype is limited to 20 characters by constraining the autoencoder generation process. The fitness function is also changed so that genotypes longer than 20 characters are assigned a negative fitness to prevent the native encoding genetic operators from omitting this limitation. The latent representations are learned for the *f1* and *f9* native encodings. Their length distributions are shown in Fig. 6. The *f1*

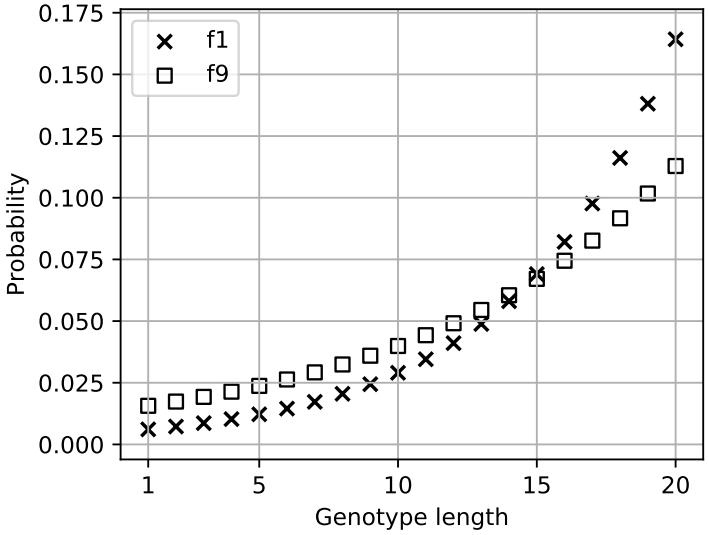

Figure 6: Genotype length distributions for native *f1* and *f9* encodings.

Table 1: Comparison of settings for native encodings.

Encoding	Dictionary	Total tokens (+*start*/*stop*)	Distribution weight formula	Token embedding dimensions
f1	`X(),RrLl`	10	$2^{0.25 \cdot length}$	7
f9	`LRDUBF`	8	$2^{0.15 \cdot length}$	5

grammar is reduced to 4 base symbols and 4 modifiers, which allows branching, branching rotation, and sticks length modification. The parameters of the approaches are compared in Table 1.

The employed architecture, chosen experimentally, is a two-layer, bidirectional autoencoder with regularization of the latent activity and network parameters with the $2 \cdot 10^{-6}$ weight. It uses the final hidden state h of the encoder for the placeholder signal in order to repeat h for every time step. The hidden state of the decoder is initialized with constant values (instead of h) to facilitate generation of the start token at the beginning of the output sequence.

LSTMs have 64 cells. The number of the encoder LSTM cells is equal to the decoder LSTM cells: $C_E = C_D = 64$, which yields a 128-dimensional latent space. Three versions of autoencoders are considered: with reconstruction loss (*reconstr*), with reconstruction and phenotype locality loss (*reconstr+pheno*), and with reconstruction and fitness locality loss (*reconstr+fitness*). The locality weight is empirically set to 3.

Baseline results include optimizing in the latent space of an untrained (random) autoencoder and evolution using the *reconstr* autoencoder with little to no selection pressure (only elitism present). The results for *f1* and *f9* are shown in Fig. 7. For the *f9* encoding and nearly no selection pressure, training the autoencoder increases the expected fitness of solutions. In the case of the *f1* base encoding, many genotype sequences generated by autoencoders may be invalid; invalid genotypes are assigned the fitness of -1, which lowers the mean best fitness as demonstrated in the *f1* (left) plot.

Fig. 8 shows fitness-distance correlation comparison of sampled points and their representation specific equivalents (*slp*) for *f1* and *f9* models. In general, fitness-distance correlation is stronger for sequence-latent points than for sampled points. In the case of *f9*, FDC is higher (coefficient is lower) for *reconstr+fitness* model and lower for *reconstr+pheno* model. Conversely, in the case of *f1*, FDC is higher (coefficient is lower) for *reconstr+pheno* model and significantly lower for *reconstr+fitness* model.

Fig. 9 shows the distribution of the fitness of the best evolved structures using the *f1* and *f9* base encoding. In the case of ordinary latent operators and the *f9* base encoding, all models perform similarly and the sequence-latent point (SLP) operators improve the results of ordinary operators to a similar degree. For the ordinary latent operators and the *f1* base encoding, all three models also perform similarly. However, in the case of the SLP operators and *f1*, the *reconstr+fitness* model is significantly worse. Unlike in the case of *f9*, in *f1* the SLPs do not improve the fitness value.

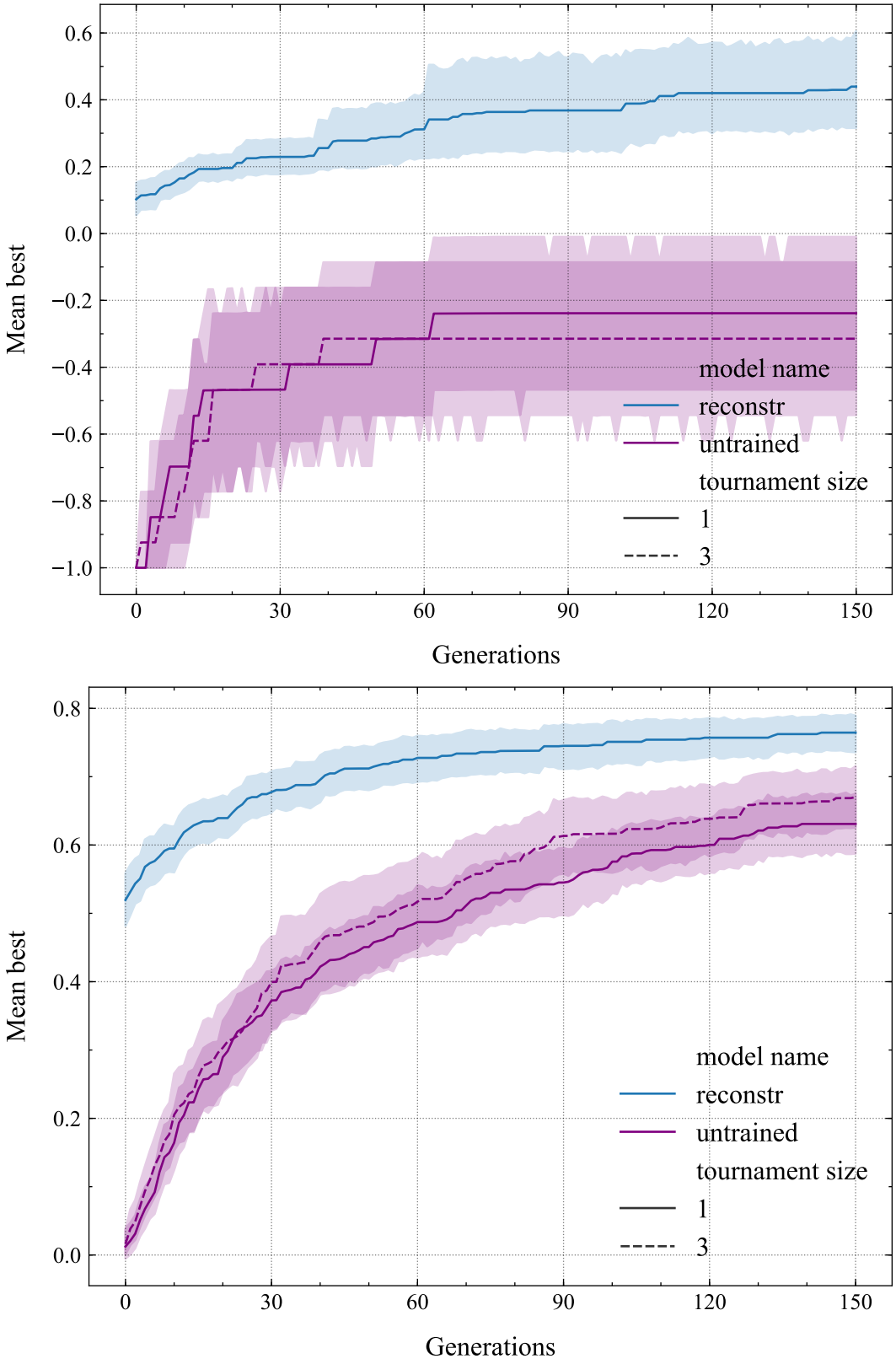

Figure 7: Baseline results (first 150 generations out of total 250) for *f1* (left) and *f9* (right) native encoding: mean best of population fitness per generation in the evolution. 95% confidence intervals are shaded.

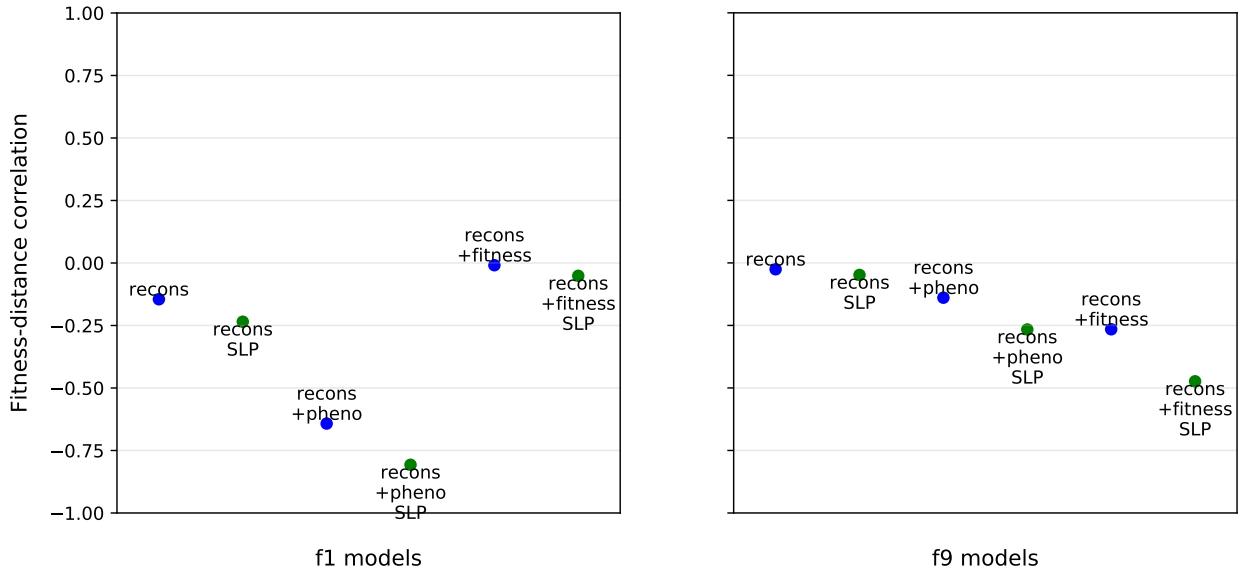

Figure 8: Comparison of fitness–distance correlation of sampled points and their SLP equivalents for *f1* (left) and *f9* (right) models.

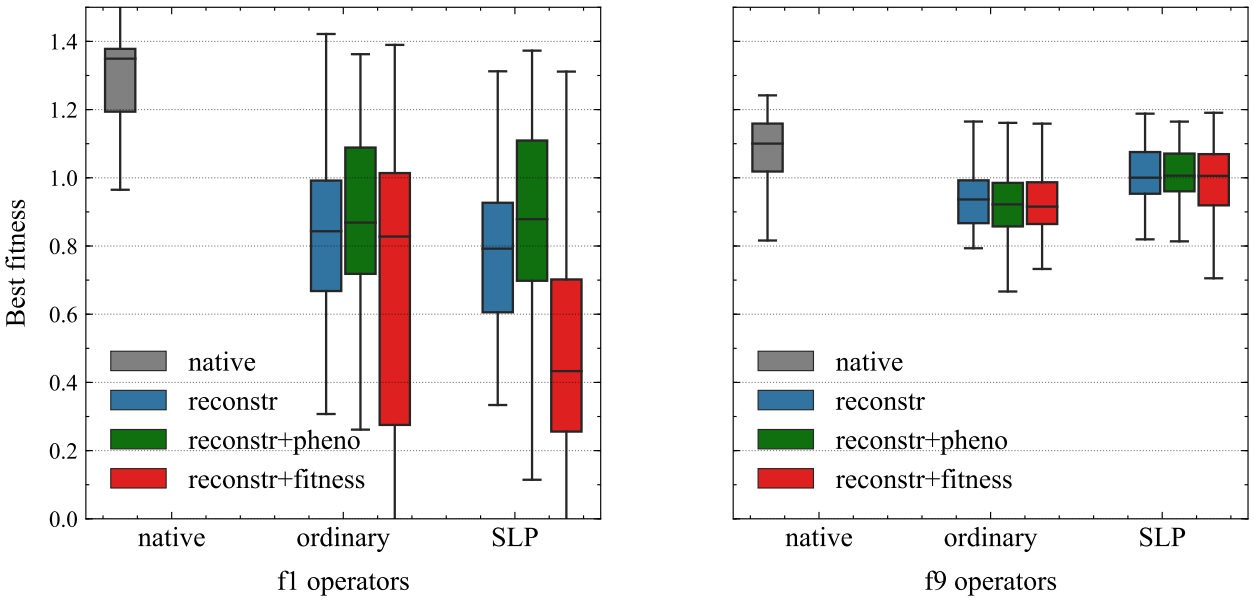

Figure 9: Fitness of the best evolved structures for the *f1* (left) and *f9* (right) base encoding. Each box shows the distribution of 130 best fitness values, each value resulting from an independent evolutionary run. Horizontal lines inside the boxes represent the medians.

4 Conclusions

Properties of the latent representation investigated before optimization suggested that they may be useful and provide a competitive level of evolvability. Fitness-distance correlation analysis indicated that models with *locality loss* may perform well in optimization. Genetic operators based on SLPs looked more promising as they generally increased the correlation. The *reconstr+fitness* model for *f1* behaved unexpectedly – it showed near-zero FDC. Overall, the optimization in the *f1* native encoding seemed to be harder given all the properties that were inspected. Operators based on SLPs improved evolution results for the *f9* native encoding for all models. The worst latent representation for *f1* turned out to be the *reconstr+fitness* model, which is consistent with the FDC predictions of problem difficulty.

Based on the conducted experiments, the latent representation with its operators cannot be considered better than native representations with their basic, yet effective operators on sequences. The latent representations seem to prefer easier optimization problems from the perspective of the native encoding and to work better in genotype spaces with a relatively high density of well-scoring solutions. Overall, the learned representation demonstrated a number of advantages when compared to manually designed encodings, even though native representations and their dedicated operators yielded higher fitness as shown in Fig. 9. However, the evolution of individuals encoded in native representations suffers from the problem of the initial plateau – if the initial population contains only the simplest genotypes, many random mutations must accumulate in order to find the first phenotype with non-zero fitness (i.e., a non-flat 3D structure), which takes many generations depending on the cumulated probabilities of mutation events and their bias. To overcome this problem which did not exist in latent representations, the initial population was seeded not with the simplest genotype in a given native encoding, but with a random genotype from the set of genotypes used for training the autoencoder.

The learned latent representation, contrary to its manually created counterpart, is scalable (more complex autoencoders yielded better results) and the methodology described here does not need to change when the autoencoder logic and the amount of data (the set of genotypes) is scaled up. The process of representation development is fully automated, while the manual development of encodings, especially as complex as *f1*, often requires many years of fixes and improvements. The topology of the new latent space of solutions can be easily modified by influencing the learning process according to the given needs and criteria, and can provide different search trajectories than a manually designed encoding and operators. As the latent representation is numerical and continuous, it facilitates the search using various gradient algorithms and more sophisticated search strategies like CMA-ES [3]. Finally, the aforementioned advantages of the automated development of the latent representation were displayed even though it inherited the limitations of the underlying base encodings, because all latent genotypes were ultimately translated to *f1* or *f9*.

4.1 Future work

There are many potential variations and extensions to the architecture presented in this paper, including modified training schemes, additional cost function terms, and smarter search operators. Another improvement would be to use a graph representing a 3D structure as the input and the output of the autoencoder. Such a direct phenotypic input/output representation could overcome biases introduced by the sequential genetic encodings.

The low density of high-fitness solutions is often a problem in evolutionary design, as demonstrated by the vertical position optimization for the *f1* native encoding. An explicit modification that addresses learning features of high-fitness solutions could increase the probability of these features appearing in solutions generated by the decoder. A simple example is an augmentation of the training set to increase the ratio of well-scoring solutions by randomly adding a few of them to every training batch, or even constructing the entire training set from local optima found by some optimization algorithm. Another proposed approach is to learn the latent representation online

(incrementally) while performing optimization, based on the solutions from the neighborhood of the current population, to smoothen the landscape in the vicinity of the population.

Fixed dimensionality of the latent representation can lead to better feature generalization but can also be a limiting factor even though the space is continuous. The following study could examine the effects of dimensionality reduction by experimenting with different sizes of the latent representation. Important aspects of this analysis could include the generalization level of mutation operator in the latent space with respect to the native representation and its optimization results. It would be interesting to extend the standard autoencoder architecture of fixed dimensionality to allow for its adaptive scaling (i.e., a variable length latent representation) in order to better handle 3D structures that inherently possess variable complexity.

A sequence-latent point (SLP) was defined as a point *slp* produced by encoding the decoded sequence corresponding to some latent point p. An extension that could improve the arrangement of solutions and thus the fitness landscape is to learn to close the gap between p and *slp* in the latent space. This could be done by introducing a loss function that would minimize the distance between p and *slp*, and introducing a different training scheme. In such a training scheme, points would be sampled in the latent space and passed through the decoder first, and then through the encoder to produce SLPs.

References

[1] Peter Bentley and Jonathan Wakefield. Generic evolutionary design. In *Soft computing in engineering design and manufacturing*, pages 289–298. Springer, 1998.

[2] Félix-Antoine Fortin, François-Michel De Rainville, Marc-André Gardner, Marc Parizeau, and Christian Gagné. DEAP documentation, 2021. URL: `https://deap.readthedocs.io`.

[3] Nikolaus Hansen. The CMA evolution strategy: A tutorial. *arXiv preprint arXiv:1604.00772*, 2016.

[4] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the 2015 IEEE International Conference on Computer Vision (ICCV)*, ICCV '15, page 1026–1034, USA, 2015. IEEE Computer Society. `doi:10.1109/ICCV.2015.123`.

[5] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[6] Gregory Hornby and Jordan Pollack. Body-brain co-evolution using L-systems as a generative encoding. In *Proceedings of the 3rd Annual Conference on Genetic and Evolutionary Computation*, pages 868–875. Morgan Kaufmann Publishers Inc., 2001.

[7] Gregory Hornby and Jordan Pollack. Creating high-level components with a generative representation for body-brain evolution. *Artificial life*, 8(3):223–246, 2002.

[8] Terry Jones and Stephanie Forrest. Fitness distance correlation as a measure of problem difficulty for genetic algorithms. In *Proceedings of the 6th International Conference on Genetic Algorithms*, pages 184–192, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc.

[9] Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 12 2014.

[10] Maciej Komosinski and Agnieszka Mensfelt. A flexible dissimilarity measure for active and passive 3D structures and its application in the fitness–distance analysis. In Paul Kaufmann and Pedro A. Castillo, editors, *Lecture Notes in Computer Science, Applications of Evolutionary Computation*, volume 11454, pages 106–121. Springer, 2019. `doi:10.1007/978-3-030-16692-2_8`.

[11] Maciej Komosinski and Adam Rotaru-Varga. Comparison of different genotype encodings for simulated 3D agents. *Artificial Life Journal*, 7(4):395–418, Fall 2001. doi:10.1162/106454601317297022.

[12] Maciej Komosinski and Szymon Ulatowski. Framsticks: Creating and understanding complexity of life. In Maciej Komosinski and Andrew Adamatzky, editors, *Artificial Life Models in Software*, chapter 5, pages 107–148. Springer, London, 2nd edition, 2009.

[13] Maciej Komosinski and Szymon Ulatowski. Framsticks website, 2021. URL: http://www.framsticks.com.

[14] Maciej Komosinski and Szymon Ulatowski. Genetic encodings in Framsticks, 2021. URL: http://www.framsticks.com/a/al_genotype.

[15] Maciej Komosinski and Szymon Ulatowski. Python interface for Framsticks, 2021. URL: http://www.framsticks.com/trac/framsticks/browser/framspy.

[16] Paweł Liskowski, Krzysztof Krawiec, Nihat Engin Toklu, and Jerry Swan. Program synthesis as latent continuous optimization: Evolutionary search in neural embeddings. In *Proceedings of the 2020 Genetic and Evolutionary Computation Conference*, GECCO '20, page 359–367, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3377930.3390213.

[17] Franz Rothlauf. Chapter 3.3: Locality. In *Representations for genetic and evolutionary algorithms*, pages 73–95. Springer-Verlag, 2006.

[18] David Rumelhart, Geoffrey Hinton, and Ronald Williams. Learning internal representations by error propagation. Technical report, California Univ San Diego La Jolla Inst for Cognitive Science, 1985.

[19] Mayu Sakurada and Takehisa Yairi. Anomaly detection using autoencoders with nonlinear dimensionality reduction. In *Proceedings of the MLSDA 2014 2nd Workshop on Machine Learning for Sensory Data Analysis*, MLSDA'14, page 4–11, New York, NY, USA, 2014. Association for Computing Machinery. doi:10.1145/2689746.2689747.

[20] Kenneth Stanley. Compositional pattern producing networks: A novel abstraction of development. *Genetic programming and evolvable machines*, 8(2):131–162, 2007.

[21] Vedat Toğan and Ayşe Daloğlu. Optimization of 3D trusses with adaptive approach in genetic algorithms. *Engineering Structures*, 28(7):1019–1027, 2006.