Neural Network Architecture Search with Differentiable Cartesian Genetic Programming for Regression

Marcus Märtens
European Space Agency
Noordwijk, The Netherlands
marcus.maertens@esa.int

Dario Izzo
European Space Agency
Noordwijk, The Netherlands
dario.izzo@esa.int

ABSTRACT

The ability to design complex neural network architectures which enable effective training by stochastic gradient descent has been the key for many achievements in the field of deep learning. However, developing such architectures remains a challenging and resource-intensive process full of trial-and-error iterations. All in all, the relation between the network topology and its ability to model the data remains poorly understood. We propose to encode neural networks with a differentiable variant of Cartesian Genetic Programming (dCGPANN) and present a memetic algorithm for architecture design: local searches with gradient descent learn the network parameters while evolutionary operators act on the dCGPANN genes shaping the network architecture towards faster learning. Studying a particular instance of such a learning scheme, we are able to improve the starting feed forward topology by learning how to rewire and prune links, adapt activation functions and introduce skip connections for chosen regression tasks. The evolved network architectures require less space for network parameters and reach, given the same amount of time, a significantly lower error on average.

KEYWORDS

designing neural network architectures, evolution, genetic programming, artificial neural networks

1 INTRODUCTION

The ambition of artificial intelligence (AI) is to develop artificial systems that exhibit a level of intelligent behaviour competitive with humans. It is thus natural that many research in AI has taken inspiration from the human brain [11]. The general brain was shaped by natural evolution to give its owner the ability to learn: new skills and knowledge are acquired during lifetime due to exposure to different environments and situations. This lifelong learning is in stark contrast to the machine learning approach, where typically only weight parameters of a static network architecture are tuned during a training phase and then left frozen to perform a particular task.

While exact mechanisms in the human brain are poorly understood, there is evidence that a process called *neuroplasticity* [5] plays an important role, which is described as the ability of neurons to change their behaviour (function, connectivity patterns, etc.) due to exposure to the environment [25]. These changes manifest themselves as alterations of the physical and chemical structures of the nervous system.

Inspired by the idea of the neuroplastic brain, we propose a differentiable version of Cartesian Genetic Programming (CGP) [19] as a direct encoding of artificial neural networks (ANN), which we call dCGPANN. Due to an efficient automated backward differentiation, the loss gradient of a dCGPANN can be obtained during fitness evaluation with only a negligible computational overhead. Instead of ignoring the gradient information, we propose a memetic algorithm that adapts the weights and biases of the dCGPANN by backpropagation. The performance in learning is then used as a selective force for evolution to incrementally improve the network architecture. We trigger these improvements by mutations on the neural connections (rewirings) and the activation functions of individual neurons, which allows us to navigate the vast design space of neural network architectures up to a predetermined maximum size.

To evaluate the performance of our approach, we evolve network architectures for a series of small-scale regression problems. Given the same canonical feed forward neural network as a starting point for each individual challenge, we show how complex architectures for improved learning can be evolved without human intervention.

The remainder of this work is organized as follows: Section 2 relates our contribution to other work in the field of architecture search and CGP applied to artificial neural networks. Section 3 gives some background on CGP, introduces the dCGPANN encoding and explains how its weights can be trained efficiently. In Section 4 we outline our experiments and describe the evolutionary algorithm together with our test problems. Results are presented in Section 5 and we conclude with a discussion on the benefits of the evolved architectures in Section 6.

2 RELATED WORK

This section briefly explains how this work is related to ongoing research like genetic programming, neural network architecture search, neuro-evolution, meta-learning and similar.

2.1 Cartesian Genetic Programming

In its original form, CGP [19] has been deployed to various applications, including the evolution of robotic controllers [10], digital filters [18], computational art [1] and large scale digital circuits [36]. The idea to use the CGP-encoding to represent neural networks goes back to works of Turner and Miller [35] and Khan et al. [16], who coined the term CGPANN. In these works, the network parameters (mainly weights as no biases were introduced) are evolved by genetic operators, and the developed techniques are thus applied to tasks where gradient information is not available (e.g. reinforcement learning). In contrast, our work will make explicit use of the gradient information for adapting weights and node biases, effectively creating a memetic algorithm [20]. There exists some work on the exploitation of low-order differentials to learn parameters for

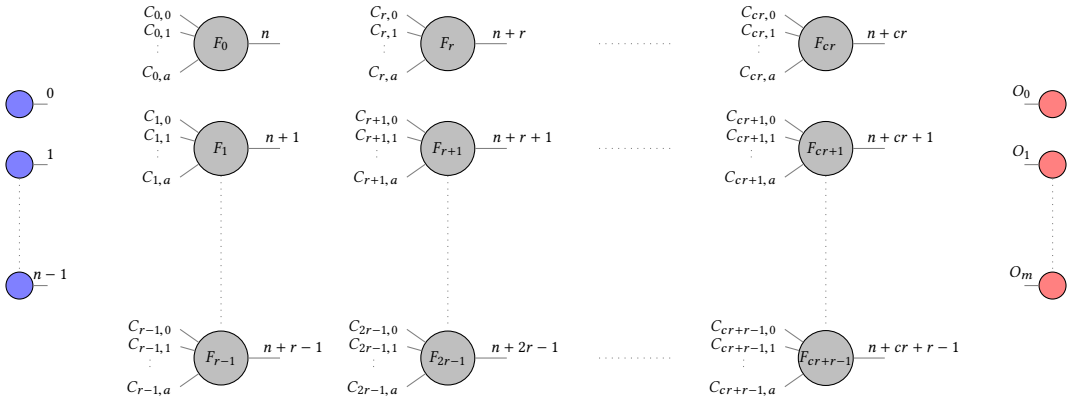


Figure 1: Most widely used form of Cartesian genetic programming, as described by [19].

genetic programs in general [6, 34] but the application of gradient descent to CGPANNs is widely unexplored.

A notable exception is the recent work of Suganuma et al. [33], who deployed CGP to encode the interconnections among functional blocks of a convolutional neural network. In [33], the nodes of the CGP represent highly functional modules such as convolutional blocks, tensor concatenation and similar operations. The resulting convolutional neural networks are then trained by stochastic gradient descent. In contrast, Our approach works directly on the fundamental units of computation, i.e. neurons, activation functions and their connectome.

2.2 Neural Network Architecture Search

There is great interest to automate the design process of neural networks, since finding the best performing topologies by human experts is often viewed as a laborious and tedious process. Some recent approaches deploy Bayessian optimization [27] or reinforcement learning [2, 38] to discover architectures that can rival human designs for large image classification problems like CIFAR-10 or CIFAR-100. However, automated architecture design often comes with heavy ressource requirements and many works are dedicated to mitigate this issue [4, 7, 26].

One way to perform architecture search are metaheuristics and neuro-evolution, which have been studied since decades and remain a profilific area of research [22]. Most notably, NEAT [32] and its variations [17, 31] have been investigated as methods to grow network structures while simultaneously evolving their corresponding weights. The approach by Han et al. [9] is almost orthogonal, as it deploys an effective pruning strategy to learns weights and topology purely from gradients. Our approach takes aspects of both: weights are learned from gradients while network topologies are gradually improved by evolution.

We focus on small-scale regression problems and optimize our topologies for efficient training as a means to combat the exploding resource requirements. In this sense, our approach is related to the concept of meta-learning [37]; the ability to "learn how to learn" by exploiting meta-knowledge and adapting to the learning task at hand. The idea to evolve effective learning systems in form of neural networks during their training has recently been surveyed by Soltoggio et al. [28], who coin the term "EPANN" (Evolved Plastic Artificial Neural Network). However, to the best of our knowledge, our work is the first to analyze plasticity in neural networks represented as CGPs.

3 DIFFERENTIABLE CARTESIAN GENETIC PROGRAMMING

This section outlines our design of a neural network as a CGP and explains how it can be trained efficiently.

3.1 Definition of a dCGPANN

A Cartesian genetic program [19], in the widely used form depicted in Figure 1, is defined by the number of inputs n, the number of outputs m, the number of rows r, the number of columns c, the levels-back l, the arity a of its kernels (non-linearities) and the set of possible kernel functions. With reference to the picture, each of the $n + rc$ nodes in a CGP is thus assigned a unique id and the vector of integers:

$$\mathbf{x}_I = [F_0, C_{0,0}, C_{0,1}, ..., C_{0,a}, F_1, C_{1,0}, ..., O_1, O_2, ..., O_m]$$

defines entirely the value of the terminal nodes. Indicating the numerical value of the output of the generic CGP node having id i with the symbol N_i, we formally have that:

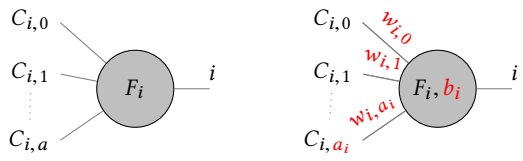
$$N_i = F_i\left(N_{C_{i,0}}, N_{C_{i,1}}, ..., N_{C_{i,a}}\right)$$

In other words, each node outputs the value of its kernel – or non linearity, to adopt a terminology more used in ANN research – computed using as inputs the connected nodes.

We modify the standard CGP node adding a weight w for each connection C, a bias b for each function F and a different arity a for each node. We also change the definition of N_i to:

$$N_i = F_i\left(\sum_{j=0}^{a_i} w_{i,j} N_{C_{i,j}} + b_j\right) \tag{1}$$

forcing the non linearities to act on the biased sum of their weighted inputs. Note that, by doing so, changing the function arity corresponds to adding or removing terms from the sum. The difference between a standard CGP node and a dCGP is depicted in Figure 2.



a) CGP b) dCGPANN

Figure 2: Differences between the i-th node in a CGP expression and in a dCGPANN expression.

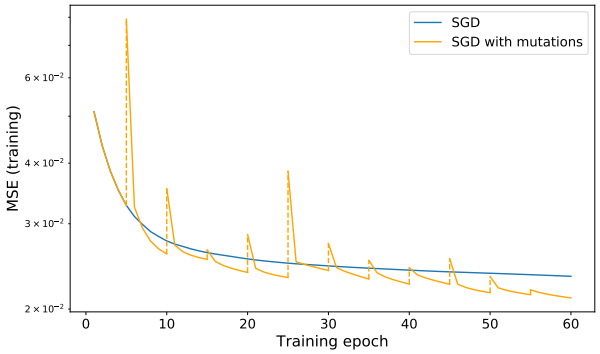


Figure 3: Loss during a generic regression task: SGD is compared with SGD perturbed, each 5 epochs, by a random (cumulative) mutation.

We define $\mathbf{x}_R$ as the vector of real numbers:

$$\mathbf{x}_R = \theta = [b_0, w_{0,0}, w_{0,1}, ..., w_{0,a_0}, b_1, w_{1,0}, w_{1,1}, ..., w_{1,a_1}, ...]$$

The two vectors $\mathbf{x}_I$ and $\mathbf{x}_R$ form the chromosome of our dCGPANN and suffice for the evaluation of the terminal values $O_i, i = 1..m$. Note that we also have introduced the symbol θ to indicate $\mathbf{x}_R$ as the network parameters are often indicated with this symbol in machine learning related literature.

3.2 Training of dCGPANNs

Training dCGPANNs is more complex than training ANNs as both the continuous and the integer part of the chromosome affects the loss ℓ and have thus to be learned. The obvious advantage is that, when successful, in a dCGPANN the learning will not only result in network parameters θ adapted to the task, but also in a network topology adapted to the task, including its size, activation functions and connections.

Consider a regression task where some loss ℓ is defined. Assume the integer part of the chromosome $\mathbf{x}_I$ is known and fixed. It is possible to compute efficiently the loss gradient $\nabla_{\mathbf{x}_R}\ell = \frac{d\ell}{d\mathbf{x}_R}$ implementing a backward automated differentiation scheme over the computational graph defined by the dCGPANN. This is not a straight forward task as the computational graph of a dCGPANN is much more intricate than that of a simple ANN, but it is attainable and eventually leads to the possibility of computing the gradient with a little computational overhead independent of θ (constant complexity). Note also that if $\mathbf{x}_I$ is changed, a new backward automated differentiation scheme needs to be derived for the changed computational graph.

Once the loss gradient is computed, the classical gradient descent rule can be used to update θ:

$$\theta_{i+1} = \theta_i - lr * \nabla_\theta \ell \tag{2}$$

where lr is used to indicate the learning rate (i.e. a trust region for the first order Taylor expansion of the loss). Epoch after epoch and batch after batch, the above update rule can be used to assemble a stochastic gradient descent algorithm during which – between epochs – the integer part of the chromosome $\mathbf{x}_I$ may also change subject to different operators, typically evolutionary ones such as mutation and crossover.

Regardless of the actual details of such a learning scheme, it is of interest to study how the SGD update rule gets disturbed by applying an evolutionary operator to $\mathbf{x}_I$. In Figure 3 we show, for a generic regression task, the trend of the training error when the update rule in Eq.(2) is disturbed, each fixed number of epochs, by mutating some active gene in $\mathbf{x}_I$. The details are not important here as the observed behaviour is generally reproducible with different tasks, networks, optimizers and evolutionary operators, but it is, instead, important to note a few facts: firstly, mutations immediately degrade the loss. This is expected since the parameters are learned on a different topology before the mutation occurs. However, the loss recovers relatively fast after a few more batches of data pass through SGD. Secondly, mutations can be ultimately beneficial as after already a few epochs the error level can be lower than what one would obtain if no mutations had occurred. In other words, we can claim that genetic operators applied on $\mathbf{x}_I$ to a stochastic gradient descent learning process working on $\mathbf{x}_R$ are mostly detrimental in the short term, but may offer advantages in the long term.

With this in mind, it is clear that the possibilities to assemble one single learning scheme mixing two powerful tools – evolutionary operators and the SGD update rule – are quite vast and their success will be determined by a clever application of the genetic operators and by protecting them for some batches/epochs before either selection or removal is applied. Note how this scheme is, essentially, a memetic approach where the individual learning component corresponds to the SGD learning of θ. In this work (Section 4.2) we will propose one easy set-up for a memetic algorithm that was found to deliver most interesting results over the test cases chosen. It makes use of Lamarckian inheritance when selecting good mutations, but it also introduces a purely Darwinian mechanism (the "Forget step") that allows for a complete reset of its accumulated experiences.

The implementation details of the dCGPANN and, most importantly, of the backward automated differentiation scheme, have been released as part of the latest release of the open source project on differentiable Cartesian Genetic Programming (dCGP) [15]. No SIMD vectorization nor GPU support is available in this release as it will, instead, be the subject of the future developments. The parallelization available so far relies on multithreading and affects the parallel evaluation of the dCGPANN expression and its gradient for data within each batch during one SGD epoch. This led us, for this paper, to experiment mainly with tasks appropriate to small networks where the performances of our code is comparable, timewise,

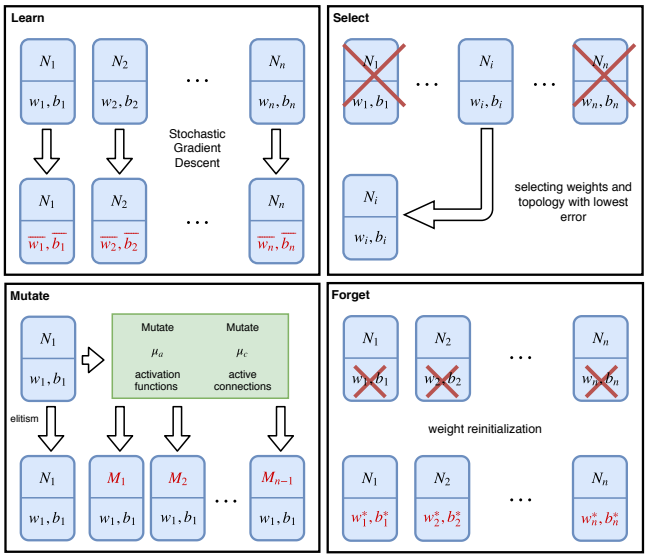


Figure 4: Each block shows one of the four operations of the LSMF algorithm. In each block, the top line corresponds to the input population of the step and the bottom line highlights the applied changes (output population).

with that of popular deep learning frameworks such as tensorflow and pytorch. Our results have thus to be taken as valid for small scale networks and their scalability to larger sizes remains to be shown and rests upon the development of an efficient, vectorized version of our current code base.

4 EXPERIMENTS

We conduct experiments to show how evolution can be applied to dCGPANNs in order to enhance their learning capabilities by continuous improvements on their network topologies. Details about the data, algorithms and our experimental setups are reported in the following.

4.1 Datasets

Our experiments are based on 6 regression problems, which we imported directly from the Penn Machine Learning Benchmarks (PMLB) [23]. We selected these problems for diversity and distinct complexity, while avoiding challenges with too many or too few instances. Table 1 shows the name of the dataset with the corresponding number of instances and features. Each problem is listed on www.openml.org where meta-data and descriptions can be found.

The features of every dataset are standardized and the single target value minmax-scaled to the unit-interval. Data is split 75/25 for training/testing.

4.2 The LSMF Algorithm

The LSMF algorithm (short for "Learn, Select, Mutate and Forget") works in J iterations, each consisting of K cycles. A cycle represents a short period of learning for a population of N dCGPANNs followed by a mutation of the best performing network. The steps that are executed during one cycle are the following:

name	#instances	#features
197_cpu_act	8192	21
225_puma8NH	8192	8
344_mv	40768	10
503_wind	6574	14
564_fried	40768	10
1201_BNG_breastTumor	116640	9

Table 1: Regression problems selected for experiments.

1. **Learn**
   For each of the N dCGPANNs, run C epochs of stochastic gradient descent with learning rate lr and mini-batches of size mb. This step only changes the $\mathbf{x}_R$ part of the chromosomes, keeping the topology of the networks $\mathbf{x}_I$ constant.
2. **Select**
   For each dCGPANN, compute the loss ℓ (i.e. training error) and select the best individual from the population (minimum ℓ), eliminating all others.
3. **Mutate**
   From the selected dCGPANN, create $N-1$ new networks by mutating a fraction of μ_a active function genes and a fraction of μ_c active connectivity genes. Elitism ensures that the new population contains the unaltered original next to its $N-1$ mutants. Complementary to Learn, this step operates only on $\mathbf{x}_I$, but leaves $\mathbf{x}_R$ unchanged.

Each cycle represents the lifelong learning of the dCGPANNs: after alterations to their network topologies, mutants are protected for C epochs during the train step to enable their development during training (see Figure 3). We call this the *cooldown period*. Eventually, the weights and biases of the dCGPANNs will converge towards a (near-)optimal loss, regardless of their current network topologies. In this situation, it becomes increasingly difficult for mutants to achieve significant improvements (as the loss is close to optimal) and the development of the topologies stalls. To resolve this situation, we execute a fourth step after K cycles, that reinitializes all weights and biases:

4. **Forget**
   For each dCGPANN, reinitialize all weights and biases while maintaining the network topology. Thus, $\mathbf{x}_R$ is randomized while $\mathbf{x}_I$ remains unchanged.

After the Forget step, a new evolutionary iteration begins, consisting of another K cycles of Learn, Select and Mutate. The result of each evolutionary iteration is a new dCGPANN network topology, which can be extracted from the last performed Select step. Figure 4 illustrates the four different steps inside LSMF and shows how each manipulates the population of dCGPANNs.

4.3 Experimental Setup and Hyperparameters

The LSMF algorithm needs a set of hyperparameters, which can be tuned to adapt to specific problems. Thus, to show the true potential of LSMF, we would need to sample the hyperparameter space for each problem separately and pick the best combination of values, which is beyond the scope of this work. Instead, we want to highlight the broad applicability and general robustness that comes

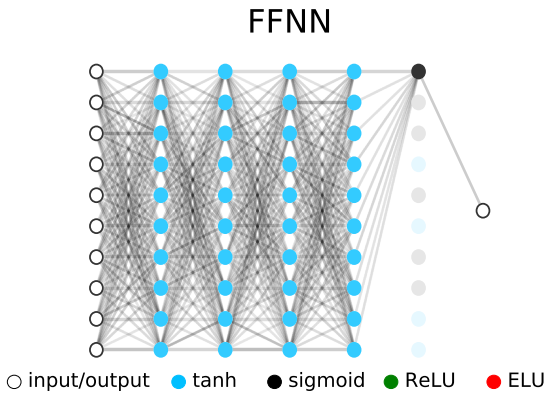


Figure 5: Template network for a problem with 10 input features and one output. Number of input nodes will vary with the problem.

dCGP parameters	
rows r	10
columns c	4
arity first column a_1	number of inputs
arity other columns a_2, a_3, a_4	10
levels-back l	3
set of kernels	`tanh`, `sig`, `ReLU`, `ELU`
Evolutionary Parameters	
population N	100
mutation rate functions μ_a	0.02
mutation rate connections μ_c	0.01
cooldown period C	1 epoch
number of cycles per iteration K	30
evolutionary iterations J	10
Parameters for Learning	
learning rate lr	0.01
batch size mb	10
initialization biases	0
initialization weights	$\mathcal{N}(0, 1)$

Table 2: Set of hyper-parameters for the experiments.

with genetic algorithms like LSMF over multiple regression problems by a reasonable choice of problem-independent fixed values. Table 2 lists our selection of hyperparameters. In the following, we elaborate on some of those choices and provide more details on the experimental setup.

In our experiments, the LSMF algorithm operates on a population of $N = 100$ dCGPANNs, which we initialize as feed-forward neural networks with 4 hidden layers, consisting of 10 neurons of activation function `tanh` each. The single output neuron uses a `sig` activation function. The set of possible kernels consists of `tanh`, `sig`, `ReLU` and `ELU`. Figure 5 shows this architecture, which we call the *template network* in the following. Preliminary experiments showed that the template network is able to solve all tasks at hand and provides a good starting point for mutations. Note, that while in the template network only neighboring layers are connected, mutations can cause rewirings to *skip* some of its layers. In particular, we set the levels-back parameter of its dCGPANN representation to 3, allowing a connection to bypass at most two layers.

Because our population size is $N = 100$, each Mutate step will generate 99 mutants from the selected best dCGPANN. Each mutation changes 1% of the connection genes and 2% of the activation function genes uniformly at random within their respective bounds. Given the fixed structure of our template network, this results in 5 to 7 mutated connections (depending on the number of inputs) and 1 mutated activation function during each Mutate step for the upcoming experiments.

Although all template networks have the same topology, their initial weights and biases are different. In particular, weights are drawn from the standard normal distribution while biases are initialized with zero. Whenever stochastic gradient descent is applied, this happens with a fixed learning rate of 0.1 and a mini-batch size of 10. The cooldown period C (i.e. the time mutants are trained before selection) is set to 1 epoch, i.e. a full pass over the training data. For large datasets with millions of instances, a whole epoch might be unnecessarily long, but we found this value to work out for the small-scale data that we analyze.

We execute $K = 30$ cycles of Learn - Select - Mutate for each evolutionary iterations. In total, we perform $J = 10$ evolutionary iterations and track the development of the dCGPANN topology at the end of each.

5 RESULTS

Our experiments show that LSMF is able to lower the average test error after each evolutionary iterations for all selected problems. In Figure 7 we show the test error for each evolutionary run averaged over a population of 100 different random weight initializations. Each solid line corresponds to an unperturbed training (no mutations) of the best evolved topology together at the end of the previous iteration (and the template network for the first iteration).

To analyze if LSMF has any selective pressure to drive optimization towards better learning or simply amounts to some form of random search, we also visualize the average test error of 100 random dCGPANNs, removing 5% outlier. A random dCGPANN is generated by drawing random numbers uniformly for all genes within their corresponding bounds and constraints of $\mathbf{x_I}$ and by initializing $\mathbf{x_R}$ in the same way as non-random dCGPANNs (zero mean, normally distributed weights).

It turns out that after at most six evolutionary iterations, LSMF has evolved topologies which perform better than random dCGPANNs on average. The difference between the test error of the randoms networks in comparison with the test error of the best evolved topology is significant with $p < 0.05$ according to separate Wilcoxon Rank sum tests for each regression problem. Remarkably, the random dCGPANNs seem to perform (on average) still better than the initial feed-forward neural networks. The (average) performance of the random dCGPANNs is shown as dashed black line in Figure 7. Numerical values for training and test error of dCGPANNs and the test error of the random dCGPANNs are shown in Table 4.

Figure 8 shows an example of an evolving topology by depicting the network structure of the dCGPANN with lowest training error

name	#weights	#active weights	#skip connections	#duplicate connections	compression ratio
197_cpu_act	520	480	78	98	0.734615
225_puma8NH	390	330	90	97	0.597436
344_mv	410	370	85	95	0.670732
503_wind	450	370	132	108	0.582222
564_fried	410	370	81	91	0.680488
1201_BNG_breastTumor	400	350	127	98	0.630000

Table 3: Summary on different connection types in the best evolved dCGPANNs. The column "#weights" describes the maximum number of weights as used by the initial feed forward neural network. The compression ratio describes by what fraction this number can be reduced in the evolved topology because of either inactive or duplicate connections.

name	evolved dCGPANN training error	evolved dCGPANN test error	random dCGPANN test error
197_cpu_act	1.851e-03 (±3.925e-04)	2.071e-03 (±4.599e-04)	2.921e-03 (±1.748e-03)
225_puma8NH	2.104e-02 (±1.428e-03)	2.031e-02 (±1.392e-03)	2.364e-02 (±3.420e-03)
344_mv	3.372e-04 (±1.571e-04)	3.657e-04 (±1.555e-04)	5.655e-04 (±2.457e-04)
503_wind	6.799e-03 (±4.366e-04)	7.945e-03 (±4.631e-04)	8.690e-03 (±8.353e-04)
564_fried	2.176e-03 (±5.324e-04)	2.878e-03 (±6.355e-04)	4.181e-03 (±1.622e-03)
1201_BNG_breastTumor	2.016e-02 (±2.348e-04)	2.345e-02 (±5.472e-04)	2.436e-02 (±7.596e-03)

Table 4: Numerical values for mean and standard deviation of training and test error for random and evolved dCGPANNs.

at the end of each corresponding iteration. For this particular example, the impact of the activation function mutations is clearly visible: the prevalence of tanh nodes in the hidden layers drops from 100% down to approx. 13%, which is much lower than one would expect by a purely a random assignment of activation functions. We further observe that sigmoidal activations appear dominantly, next to ReLU and occasional ELU nodes. Moreover, the tanh activation functions of the first layer are completely substituted after the last evolutionary iteration.

Analyzing the structure of the evolved network population, we observe across all problems that certain connections are dropped while others are enforced by rewiring links on top of each other. While the dCGPANN encoding enables such k-fold links, they are redundant for computation and may be substituted by a single link containing the sum of the k connection weights.

We further observe that, on average, half of the nodes in the last layer become inactive, which further reduces the size of the network model. Only for the 503_wind problem, nodes in lower layers have been observed to become inactive as well (see Figure 6). Inactive connections and duplicate connections allow to lower the space requirements for the models by about 40%. Specific values for possible model compression are listed in Table 3.

The reason for the appearance of inactive nodes are rewired connections that go either into k-fold links or to an entirely different hidden layer. As the levels-back parameter in our experiment is 3, any of the 10 incoming connections of the output node can be rewired to *skip* the last (or even the second-last) layer, necessarily resulting in inactive nodes within the last layer. These skip connections can appear also at other places in the network, bypassing other hidden layers or feeding directly into the inputs (compare

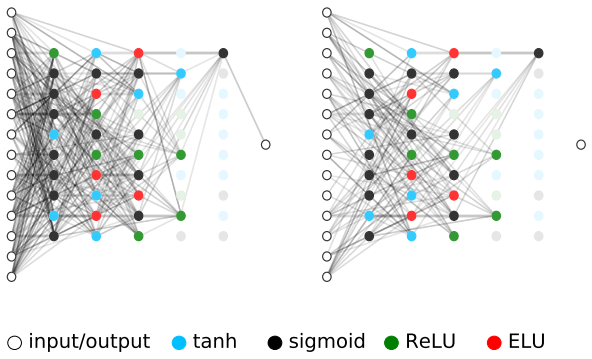


Figure 6: Final evolved topology after 10 iterations of LSMF on the problem 503_wind. Left: all active connections. Right: skip connections only (active connections who do not connect two nodes in neighboring columns).

right side of Figure 6). We report the total number of evolved skip-connections in Table 3.

6 DISCUSSIONS AND CONCLUSION

Our experiments show that it is possible to find a dCGPANN starting from a feed forward neural network that increases the speed of learning while at the same time reducing the complexity of the model for several regression problems. These two effects might not be unrelated, as smaller models are (generally) faster to train. However, while random dCGPANNs are on average even smaller than the evolved dCGPANNs, their performance falls behind after a

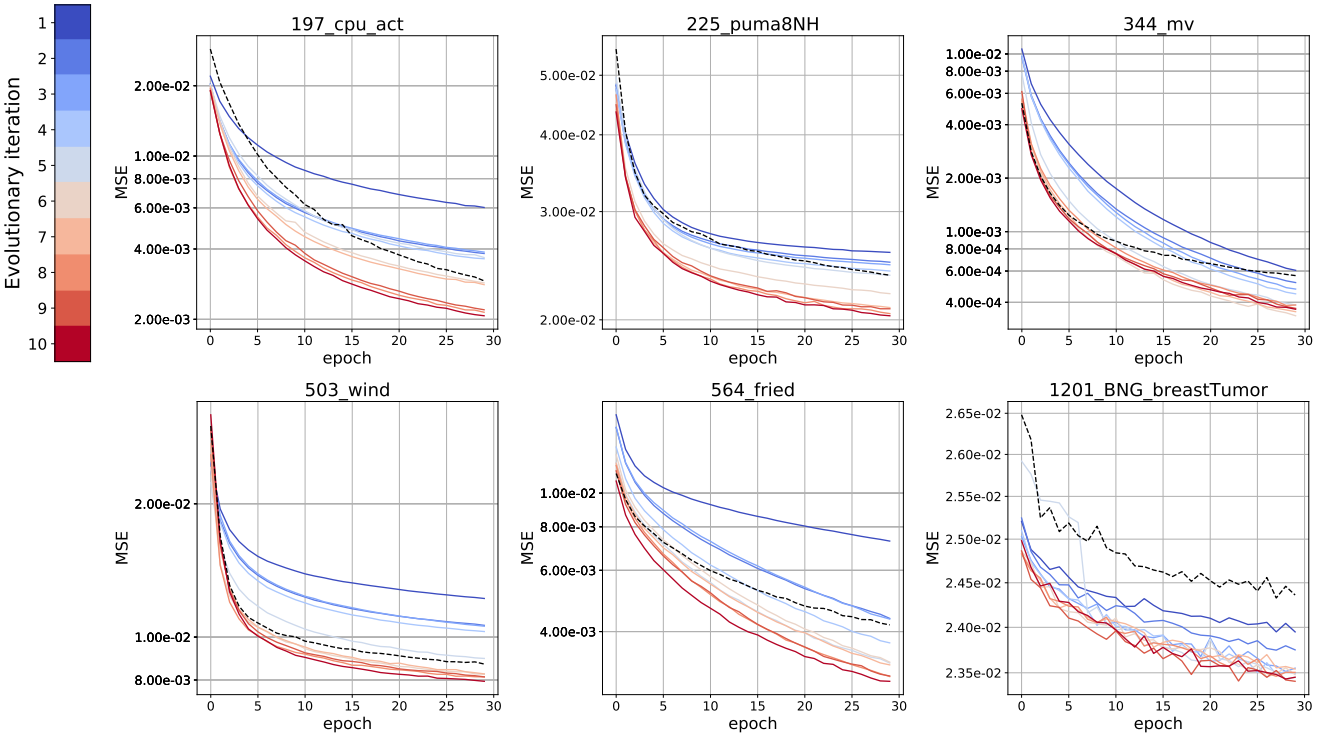


Figure 7: Learning curves for selected regression problems, starting with a 4 layer feed-forward neural network (dark blue) and applying the LSMF-algorithm for 10 iterations (last iteration: dark red). Reported is the average MSE of a population of 100 dCGPANN with different sets of starting weights in log-scale. The dashed black line corresponds to the average MSE of a population of 100 randomly generated dCGPANNs, each running with one of the 100 starting weights.

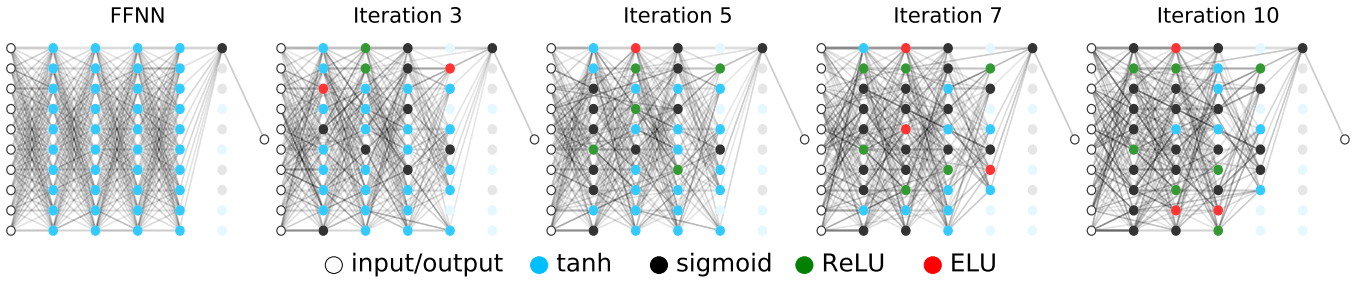


Figure 8: Evolution of dCGPANN topology by running the LSMF-algorithm on the problem 344_mv. Leftmost: initial topology (feed-forward neural network). From left to right: best performing topologies (lowest MSE at the end of an iteration) at several points during the 10 iterations of LSMF. The thickness of a connection is proportional to its weight.

couple of evolutionary iterations. This implies that LSMF-like algorithms are able to effectively explore the search space of dCGPANN topologies. Thus, there are reasons to assume that the performance of neural networks might be generally enhanced by the deployment of dCGPANNs.

For example, when comparing the evolved structures, it becomes clear that certain activation functions are selected more than others. This is reflected in recent neural network research, which acknowledges their importance [3, 21, 24]. While so far only a few vague rules and human intuition could assist in setting these functions in favor of particular problems, LSMF allows to automatically evolve beneficial activation functions on the level of individual neurons.

Another reason for the superior performance of dCGPANNs is the emergence of skip connections, which are widely acknowledged as one of the major mechanism to overcome the vanishing gradient problem [13], a major roadblock in the development of Deep Learning. Many of the modern deep architectures like ResNets [12],

HighwayNets [30] and DenseNets [14] need such skip connections to enable effective training.

A further well-known method that has been used to enhance neural network training is dropout [29]. This procedure is simulated by dCGPANN by use of inactive nodes, which are frequently appearing and disappearing due to the random rewirings of Mutate steps. Dropout is mainly used to prevent overfitting, a problem that we have not encountered in our experiments so far. However, if overfitting was an issue, a three-way split of the data would provide an easy solution as it would enable the Select step to be independent from the training and test data.

While 10 iterations of LSMF have been proven effective for most problems, there is no guarantee that the algorithm will *always* reduce the error after every iteration (In fact, the problem 344_mv in Figure 7 provides a counterexample). This convergence behavior could arise from the rigid use of hyperparameters that we deployed to show the broad applicability of LSMF. A problem-dependent adaptation of these hyperparameters between different iterations of LSMF would most likely mitigate this issue.

Due to an efficient implementation of automated differentiation, the resources which are needed to enable backpropagation in dCGPANNs are negligible. In fact, the software we deploy evaluates (on the CPU) networks with their gradient nearly 2 times faster than comparable neural network frameworks like tensorflow or pytorch on the problems that we presented here. The emergence of skip connections and dropout-like behaviour gives us reason to believe that evolution of dCGPANNs might be especially beneficial for larger and deep architectures. In particular, any pre-defined deep layered network can be transformed into a dCGPANN to serve as a starting point for structural optimization while training takes place. Also smaller dCGPANN subnetworks (i.e. last layers) can be deployed for evolution or used as buildings blocks inside automated machine learning frameworks (e.g. [8]). However, because in its current release no SIMD vectorization nor GPU support is available, analyzing larger network architectures (as they are frequently deployed for example in image classification tasks), becomes very time consuming. Thus, the scalability of our approach has to remain open until GPU-support or equivalent forms of parallelization become available for proceeding studies.

Next to matters of scalability, further research can be directed to more sophisticated genetic algorithms working on dCGPANNs. While our mutations targeted connections and activation functions, it would be of interest to study their effects separately and further develop mutation operators (potentially by incorporating gradients) that further improve the algorithm. While our choices of hyperparameters achieved already promising results, a better adaptation of the cooldown period, learning rate, mutation rates and weight initialization could potentially lead to even larger improvements. We deliberately avoided delving too deep into the impact of these hyperparameters to demonstrate the broad applicability of evolutionary optimization in combination with stochastic gradient descent for weight adaptation. Since both of these aspect can be handled efficiently if neural networks are represented as dCGPANNs, we believe that this encoding can open the door to transfer plenty of helpful and established evolutionary mechanism into network training, bringing us a step closer to achieving neural plasticity for our artificial neural networks.

REFERENCES

[1] Laurence Ashmore and Julian Francis Miller. 2003. Evolutionary Art with Cartesian Genetic Programming. *Technical Report* (2003).

[2] Bowen Baker, Otkrist Gupta, Nikhil Naik, and Ramesh Raskar. 2016. Designing Neural Network Architectures using Reinforcement Learning. arXiv:cs.LG/1611.02167

[3] Mina Basirat and Peter M. Roth. 2018. The Quest for the Golden Activation Function. arXiv:cs.NE/1808.00783

[4] Han Cai, Tianyao Chen, Weinan Zhang, Yong Yu, and Jun Wang. 2018. Efficient Architecture Search by Network Transformation. AAAI.

[5] Bogdan Draganski, Christian Gaser, Volker Busch, Gerhard Schuierer, Ulrich Bogdahn, and Arne May. 2004. Neuroplasticity: changes in grey matter induced by training. *Nature* 427, 6972 (2004), 311.

[6] Z Emigdio, Leonardo Trujillo, Oliver Schütze, Pierrick Legrand, et al. 2015. A local search approach to genetic programming for binary classification. In *Proceedings of the 2015 on Genetic and Evolutionary Computation Conference-GECCO'15*.

[7] Jiemin Fang, Yukang Chen, Xinbang Zhang, Qian Zhang, Chang Huang, Gaofeng Meng, Wenyu Liu, and Xinggang Wang. 2019. EAT-NAS: Elastic Architecture Transfer for Accelerating Large-scale Neural Architecture Search. arXiv:cs.CV/1901.05884

[8] Isabelle Guyon, Imad Chaabane, Hugo Jair Escalante, Sergio Escalera, Damir Jajetic, James Robert Lloyd, Núria Macià, Bisakha Ray, Lukasz Romaszko, Michèle Sebag, et al. 2016. A brief review of the ChaLearn AutoML challenge: any-time any-dataset learning without human intervention. In *Workshop on Automatic Machine Learning*. 21–30.

[9] Song Han, Jeff Pool, John Tran, and William Dally. 2015. Learning both weights and connections for efficient neural network. In *Advances in neural information processing systems*. 1135–1143.

[10] Simon Harding and Julian Francis Miller. 2005. Evolution of Robot Controller Using Cartesian Genetic Programming. In *European Conference on Genetic Programming*. Springer, 62–73.

[11] Demis Hassabis, Dharshan Kumaran, Christopher Summerfield, and Matthew Botvinick. 2017. Neuroscience-inspired artificial intelligence. *Neuron* 95, 2 (2017), 245–258.

[12] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.

[13] Sepp Hochreiter. 1998. The vanishing gradient problem during learning recurrent neural nets and problem solutions. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems* 6, 02 (1998), 107–116.

[14] Gao Huang, Zhuang Liu, Laurens van der Maaten, and Kilian Q. Weinberger. 2017. Densely Connected Convolutional Networks. *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (2017), 2261–2269.

[15] Dario Izzo, Francesco Biscani, and Alessio Mereta. 2017. Differentiable genetic programming. In *European Conference on Genetic Programming*. Springer, 35–51.

[16] Maryam Mahsal Khan, Arbab Masood Ahmad, Gul Muhammad Khan, and Julian F Miller. 2013. Fast learning neural networks using Cartesian genetic programming. *Neurocomputing* 121 (2013), 274–289.

[17] Risto Miikkulainen, Jason Liang, Elliot Meyerson, Aditya Rawal, Daniel Fink, Olivier Francon, Bala Raju, Hormoz Shahrzad, Arshak Navruzyan, Nigel Duffy, et al. 2019. Evolving deep neural networks. In *Artificial Intelligence in the Age of Neural Networks and Brain Computing*. Elsevier, 293–312.

[18] Julian F. Miller. 1999. Evolution of Digital Filters Using a Gate Array Model. In *Evolutionary Image Analysis, Signal Processing and Telecommunications*, Riccardo Poli, Hans-Michael Voigt, Stefano Cagnoni, David Corne, George D. Smith, and Terence C. Fogarty (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 17–30.

[19] Julian F Miller. 2011. Cartesian genetic programming. In *Cartesian Genetic Programming*. Springer, 17–34.

[20] Pablo Moscato, Carlos Cotta, and Alexandre Mendes. 2004. *Memetic Algorithms*. Springer Berlin Heidelberg, Berlin, Heidelberg, 53–85.

[21] Vinod Nair and Geoffrey E Hinton. 2010. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*. 807–814.

[22] Varun Kumar Ojha, Ajith Abraham, and Václav Snásel. 2017. Metaheuristic design of feedforward neural networks: A review of two decades of research. *Engineering Applications of Artificial Intelligence* 60 (2017), 97–116.

[23] Randal S. Olson, William La Cava, Patryk Orzechowski, Ryan J. Urbanowicz, and Jason H. Moore. 2017. PMLB: a large benchmark suite for machine learning evaluation and comparison. *BioData Mining* 10, 1 (11 Dec 2017), 36.

[24] Prajit Ramachandran, Barret Zoph, and Quoc V. Le. 2018. Searching for activation functions. In *6th International Conference on Learning Representations ICLR 2018*.

[25] Joseph P Rauschecker and Wolf Singer. 1981. The effects of early visual experience on the cat's visual cortex and their possible explanation by Hebb synapses. *The Journal of physiology* 310, 1 (1981), 215–239.

[26] Esteban Real, Sherry Moore, Andrew Selle, Saurabh Saxena, Yutaka Leon Suematsu, Jie Tan, Quoc Le, and Alex Kurakin. 2017. Large-Scale Evolution of Image Classifiers. arXiv:cs.NE/1703.01041

[27] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. 2012. Practical Bayesian Optimization of Machine Learning Algorithms. In *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger (Eds.). Curran Associates, Inc., 2951–2959.

[28] Andrea Soltoggio, Kenneth O Stanley, and Sebastian Risi. 2018. Born to learn: the inspiration, progress, and future of evolved plastic artificial neural networks. *Neural Networks* (2018).

[29] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *Journal of Machine Learning Research* 15 (2014), 1929–1958.

[30] Rupesh Kumar Srivastava, Klaus Greff, and Jürgen Schmidhuber. 2015. Highway Networks. arXiv:cs.LG/1505.00387

[31] Kenneth O Stanley, David B D'Ambrosio, and Jason Gauci. 2009. A hypercube-based encoding for evolving large-scale neural networks. *Artificial life* 15, 2 (2009), 185–212.

[32] Kenneth O Stanley and Risto Miikkulainen. 2002. Evolving neural networks through augmenting topologies. *Evolutionary computation* 10, 2 (2002), 99–127.

[33] Masanori Suganuma, Shinichi Shirakawa, and Tomoharu Nagao. 2017. A genetic programming approach to designing convolutional neural network architectures. In *Proceedings of the Genetic and Evolutionary Computation Conference*. ACM, 497–504.

[34] Alexander Topchy and William F Punch. 2001. Faster genetic programming based on local gradient search of numeric leaf values. In *Proceedings of the 3rd Annual Conference on Genetic and Evolutionary Computation*. Morgan Kaufmann Publishers Inc., 155–162.

[35] Andrew James Turner and Julian Francis Miller. 2013. Cartesian genetic programming encoded artificial neural networks: a comparison using three benchmarks. In *Proceedings of the 15th annual conference on Genetic and evolutionary computation*. ACM, 1005–1012.

[36] Zdenek Vasicek. 2015. Cartesian gp in optimization of combinational circuits with hundreds of inputs and thousands of gates. In *European Conference on Genetic Programming* (2015-01-01). Springer, Springer, 139–150.

[37] Ricardo Vilalta and Youssef Drissi. 2002. A Perspective View and Survey of Meta-Learning. *Artificial Intelligence Review* 18, 2 (01 Jun 2002), 77–95.

[38] Barret Zoph and Quoc V. Le. 2016. Neural Architecture Search with Reinforcement Learning. arXiv:cs.LG/1611.01578