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

While optimized neural network architectures are essential for effective training with gradient descent, their development remains a challenging and resource-intensive process full of trial-and-error iterations. 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.

CCS CONCEPTS

• **Computing methodologies** → **Genetic programming**; **Neural networks**;

KEYWORDS

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

ACM Reference Format:
Marcus Märtens and Dario Izzo. 2019. Neural Network Architecture Search with Differentiable Cartesian Genetic Programming for Regression. In *Genetic and Evolutionary Computation Conference Companion (GECCO '19 Companion), July 13–17, 2019, Prague, Czech Republic.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3319619.3322003

1 INTRODUCTION

We propose a differentiable version of Cartesian Genetic Programming (CGP) [1] 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. The network architectures is improved by mutations on neural connections (rewirings) and activation functions of individual neurons. We show how complex architectures can be evolved without human intervention for a series of small-scale regression problems. We adapt the standard encoding of a Cartesian genetic program [1], which is represented by a vector of integers, encoding the functions F of each node as well as the inter-node connections C:

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

Given n input nodes and a set of possible Kernel functions, in CGP the vector $\mathbf{x}_I$ entirely determines the m outputs. 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 (non linearity) 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:

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

forcing the non linearities to act on the biased sum of their weighted inputs (compare Figure 1).

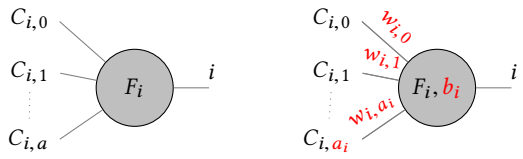

Figure 1: Node differences between CGP (left) and dCGPANN (right) expression.

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

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

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 $O_i, i = 1..m$.

2 EXPERIMENTS AND RESULTS

Our experiments are based on several regression problems, which we imported directly from the Penn Machine Learning Benchmarks (PMLB) [2], selected for diversity and distinct complexity. Due

GECCO '19 Companion, July 13–17, 2019, Prague, Czech Republic

ACM ISBN 978-1-4503-6748-6/19/07…$15.00
https://doi.org/10.1145/3319619.3322003

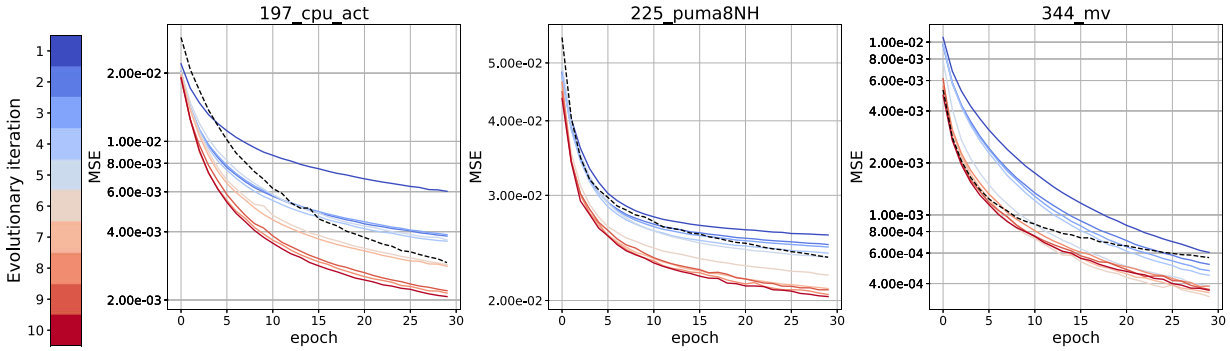

Figure 2: Learning curves for selected regression problems, starting with a 4 layer feed-forward neural network (dark blue) and applying the LSMF-algorithm for 10 iterations (last iteration: dark red). Reported is the average MSE of a population of 100 dCGPANN with different sets of starting weights 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.

to constraints in space, we only report results on the problems 197_cpu_act, 225_puma8NH and 344_mv here.

We propose the LSMF algorithm (short for "Learn, Select, Mutate and Forget") as a memetic algorithm for network design. This algorithm works in cycles, which represent short periods of learning of a population N dCGPANNs, intermixed with selection and mutation. We initialize our starting population with dCGPANN representations of feed forward neural network of 4 hidden layers (10 nodes in each with tanh activations) and a sigmoidal output neuron and begin executing cycles of Learn, Select and Mutate.

The Learn-step runs a standard stochastic gradient descent on each dCGPANN modifying only $\mathbf{x}_R$. In the Select-step, the dCGPANN with lowest training loss is selected, eliminating all others. Lastly, the elitism Mutate-step creates $N-1$ new dCGPANNs by mutating a small fraction (0.02) of active function genes and a small fraction (0.01) of active connectivity genes, modifying only $\mathbf{x}_I$. This process is repeated until the weights and biases of the dCGPANNs converge towards a (near-)optimal loss. In this situation (which appears after roughly 30 epochs for our experiments), it becomes increasingly difficult for mutants to achieve significant improvements. To resolve this situation, we execute the Forget-step which simply reinitializes all weights and biases (keeping $\mathbf{x}_I$). After the Forget step, a new evolutionary iteration begins, consisting of another 30 cycles of Learn, Select and Mutate.

We extract the evolved network topologies after each evolutionary iteration and evaluate their performance when trained unperturbed (no mutations) with stochastic gradient descent for 100 different random weight initializations. Figure 2 shows that the learning curves for the test-error decreases with each successive 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). We find 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 2.

Analyzing the structure of the evolved network population, we observe 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. Furthermore, we observe that evolution generates skip-connections, which have been crucial for the success of many modern network architectures.

3 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 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.

REFERENCES

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

[2] 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.