Evolving Node Transfer Functions in Deep Neural Networks for Pattern Recognition

Dmytro Vodianyk(✉) and Przemysław Rokita

Institute of Computer Science, Warsaw University of Technology, Nowowiejska 15/19, 00-665 Warsaw, Poland
vodianyk@gmail.com, pro@ii.pw.edu.pl

Abstract. Theoretical results suggest that in order to learn complicated functions that can represent high-level features in the computer vision field, one may need to use deep architectures. The popular choice among scientists and engineers for modeling deep architectures are feed-forward Deep Artificial Neural Networks. One of the latest research areas in this field is the evolution of Artificial Neural Networks: NeuroEvolution. This paper explores the effect of evolving a Node Transfer Function and its parameters, along with the evolution of connection weights and an architecture in Deep Neural Networks for Pattern Recognition problems. The results strongly indicate the importance of evolving Node Transfer Functions for shortening the time of training Deep Artificial Neural Networks using NeuroEvolution.

1 Introduction

Deep systems are believed to play an important role in information processing of intelligent agents. A popular hypothesis which supports this belief is that deep systems can be exponentially more efficient at representing complicated functions than their shallow counterparts [1] and for certain problems and network architectures it is proven [2]. These systems include learning methods for a wide array of deep architectures, including neural networks with many hidden layers [3]. Much attention has recently been devoted to deep neural networks, because of their theoretical appeal, inspiration from biology, and because of their empirical success in the computer vision field [4,5].

When using a backpropagation algorithm for training a feed-forward deep neural network for pattern recognition problems, one may observe that the gradient tends to get smaller as it moves backwards through the hidden layers. This means that neurons in the earlier layers learn much more slowly than neurons in the later layers. There are fundamental reasons why this happens in many real-world neural networks, and this phenomenon is known as the vanishing gradient problem [6].

Much ongoing research aims to better understand challenges that can occur when training deep networks. In 2010 Glorot and Bengio [7] found evidence that using a sigmoid transfer function can potentially cause problems in training deep neural networks. In another paper [8] Sutskever and others studied an impact on

L. Rutkowski et al. (Eds.): ICAISC 2017, Part I, LNAI 10245, pp. 206–215, 2017.
DOI: 10.1007/978-3-319-59063-9_19

deep learning of the random weight initialization and the momentum schedule in the momentum-based stochastic gradient descent.

The purpose of this research is to explore the effect of evolving a Node Transfer Function and its parameters, along with the evolution of connection weights and an architecture in Deep Neural Networks for pattern recognition problems. The paper concludes that for certain pattern recognition problems the choice of a node transfer function for each neuron is much more important than the choice of connection weights in deep neural networks. It provides a way to shorten the time of training Deep Artificial Neural Networks using the NeuroEvolutionary approach.

2 Evolution of Deep Neural Networks

Evolutionary Artificial Neural Networks (EANNs) refer to a special class of ANNs in which evolution is another fundamental form of adaptation in addition to learning [9]. One of the important features of EANNs is their adaptability to a dynamic environment. In other words, EANNs can adapt to an environment as well as changes in the environment. Evolution and learning of EANNs make adaptation to dynamic environments much more effective [10].

Evolutionary algorithms (EA) have been successfully applied for training deep neural networks [11]. Several NeuroEvolutionary systems have been successfully developed to solve various challenging tasks with remarkably better performance than traditional learning techniques [12].

The most common way to train an EANN is to evolve connection weights. It is possible to evolve a topology along with connection weights. Topology evolving methods include: GNARL [13], NEAT [14] and CGPANN [15]. EAs can be used to optimize a Node Transfer Function and its parameters of each neuron within a heterogeneous ANN. Indeed, a transfer function has been shown to be an important part of a neural network with one hidden layer [16]. Heterogeneous shallow EANNs perform much better on MNIST data when the connection weights and the transfer functions evolve simultaneously along with their parameters for each node [17]. It also was indicated that further research is required to investigate the impact of evolving a NTF along with connection weights and architecture of EANNs [17]. In this research we are taking an additional step forward – exploring the effect of evolving a NTF and its parameters, along with the evolution of connection weights and an architecture in Deep Neural Networks for Pattern Recognition problems.

The evolutionary approach to an ANN training process consists of two major phases. The first phase is to decide on the representation of connection weights and an architecture: in a form of binary strings or not. In the experiments performed in this paper, the connection weights are represented as real-number matrices. The second phase is to develop an evolutionary process, in which search operators, such as crossover and mutation, have to be defined in conjunction with the representation scheme. In this paper we use crossover and mutation operators for evolving connection weights, transfer functions and an architecture of a deep neural network. In the scope of this paper we developed an evolutionary deep system to perform tests of the proposed method on different pattern recognition problems. The evolutionary cycle of the system is illustrated on Fig. 1, where:

Decode – decoding of each individual (genotype) in the current generation into a set of connection weights, transfer functions and an architecture, constructing a corresponding Deep ANNs.
Evaluate – evaluating effectiveness of each deep neural network by computing its total mean square error between actual and target outputs. The fitness of an individual is determined by the error. The higher the error, the lower the fitness.
Select – parents selection based on their fitness for the further reproduction.
Generate – applying search operators, such as crossover and mutation for the connection weights, an architecture and transfer functions to selected parents in order to generate offspring, which form the next generation.

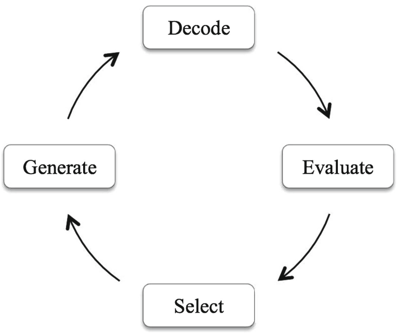

Fig. 1. An evolutionary cycle of the system.

3 Pattern Recognition Benchmarks

In order to test the proposed system and make sure that it performs better on pattern recognition problems with different training and test datasets sizes, three benchmarks were employed.

3.1 Brodatz Textures

The Brodatz Textures [18] are scans of a set of glossy black and white prints of the corresponding textures in the Brodatz book. While these prints are pictures of the same textures as in the book, in most cases they are not the same image as the one in the book. An example of the texture, which represents "Grass", can be seen on Fig. 2.

Since this is a small dataset, it was decided to use only training data for the performance validation of the deep evolutionary system. The first 26 images from the dataset were used for the experiment. Original images were resized to 30 by 30 pixels.

Fig. 2. Example of the Brodatz Texture.

3.2 Handwritten Digits

For this benchmark problem MNIST [19] data set was used. It contains tens of thousands of scanned images of handwritten digits, together with their correct classifications. MNIST's name comes from the fact that it is a modified subset of two data sets collected by NIST, the United States' National Institute of Standards and Technology. Figure 3 shows a few images from the data set.

The dataset was divided into two parts: the first part contains $1,000$ handwritten digits to be used as a training data and the second part contains $2,000$ images to be used as a test data. These images are 28 by 28 pixels in size.

Fig. 3. Examples of images from MNIST.

3.3 COIL-100

The COIL-100 database [20] contains a set of $7,200$ colored images of 100 objects on a black background. Each object in the database is represented by 72 images. Figure 4 shows an example of the image from the original dataset.

Images were resized to 20 by 20 pixels and then grayscaled using the following formula for calculating luminance [21]:

$$E_Y = 0.299 * E_R + 0.587 * E_G + 0.114 * E_B,$$

where:

E_Y – luminance of a pixel;
E_R, E_G, E_B – are red, green and blue components of a pixel.

For the training dataset it was chosen to use 40 images for each object and the other 32 images were used for the test dataset. In total $4,000$ images were used for the training dataset and $3,200$ for the test dataset.

Fig. 4. A random image from COIL-100 database.

4 Experimental Setting

4.1 Initial Architecture of the Deep Neural Network

The architecture of an artificial neural network in the experiment is not constant and can change during the evolution process. However, we still have to define an initial architecture of the neural network. It was decided to initialize an architecture of the network with one hidden layer only and see if it evolves into a deep neural network. Table 1 describes a shape of the networks used at the initial stage for each benchmark.

Table 1. Initial architectures of neural networks.

Benchmark problem	ANN shape
Brodatz Textures	$900 \rightarrow 60 \rightarrow 13$
Handwritten digits	$784 \rightarrow 60 \rightarrow 10$
COIL-100	$400 \rightarrow 60 \rightarrow 100$

It is important to notice that the evolved networks are truly deep: the number of hidden layers was varying between 4 and 5. For example, one of the evolutionary processes for COIL-100 benchmark produced a deep neural network with 4 hidden layers: $400 \rightarrow 64 \rightarrow 63 \rightarrow 69 \rightarrow 85 \rightarrow 100$.

4.2 Node Transfer Functions

The list of transfer functions was used to evolve a DNN. Table 2 shows which functions were used.

The default values of μ and σ for Unipolar Sigmoid, Sigmoid Prime, Hyperbolic and Bipolar functions are: $\mu = 0.0$, $\sigma = 1.0$ The default values of μ and σ for Gaussian function are: $\mu = 4.5$, $\sigma = 4.5$.

Table 2. List of node transfer functions.

Function name	Equation
Step	$g(x) = \begin{cases} 1, & \text{if } x \geq 0 \\ 0, & \text{if } x < 0 \end{cases}$
Unipolar Sigmoid	$g(x) = \frac{1}{1+e^{-\frac{x-\mu}{\sigma}}}$
Gaussian	$g(x) = e^{-\frac{(x-\mu)^2}{2\sigma^2}}$
Sigmoid Prime	$g(x) = \frac{e^{-\frac{x-\mu}{\sigma}}}{(1+e^{-\frac{x-\mu}{\sigma}})^2}$
Hyperbolic Sigmoid	$g(x) = \frac{e^{\frac{x-\mu}{\sigma}} - e^{-\frac{x-\mu}{\sigma}}}{e^{\frac{x-\mu}{\sigma}} + e^{-\frac{x-\mu}{\sigma}}}$
Bipolar Sigmoid	$g(x) = \frac{1-e^{-\frac{x-\mu}{\sigma}}}{e^{\frac{x-\mu}{\sigma}} + e^{-\frac{x-\mu}{\sigma}}}$

4.3 Representation of the Deep Evolutionary Neural Network

We are going to use a NeuroEvolutionary approach to train a deep neural network in our system. Therefore, we have to decide how the network should be represented and which search operators should be used for the evolutionary process.

A chromosome contains:

1. An array of matrices with connection weights which are represented as real numbers.
2. A matrix of NTFs for hidden layers and the output layer of the deep neural network.
3. A matrix with parameters which are represented as objects for node transfer functions.

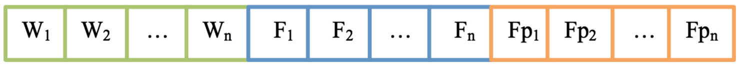

Fig. 5. Graphical representation of a chromosome.

Figure 5 shows a graphical representation of a chromosome in a DNN, where: n is a number of layers, W_1, W_2, ..., W_n are matrices of connection weights between input-hidden, hidden-hidden and hidden-output nodes; F_1, F_2, ..., F_n are arrays of transfer functions associated with each layer; Fp_1, Fp_2, ..., Fp_n are arrays of parameters for each node transfer function.

4.4 Search Operators

In the experiment we decided to use both *crossover* and *mutation* search operators to evolve a deep neural network. A population size in one generation is 100 chromosomes. The population is generated with a random number of hidden layers, random connection weights and transfer functions.

We applied a randomized version of the crossover operator. The system takes two neighboring chromosomes and makes a copy of the first one, which forms a child chromosome. Then it iterates through all layers in the second chromosome and chooses at random a connection weight and a node transfer function to be copied into the child chromosome. The number of randomly chosen connection weights and transfer functions equals to $\frac{n}{2}$, where n is the number of neurons in the layer of the current iteration.

The mutation operator applies for both current chromosomes and for chromosomes generated as the result of the crossover procedure. A probability of the mutation in the connection weights and node transfer functions is 5%.

For connection weights, a mutation operator creates a random index for each layer and changes a connection weight value between $w_k - r$ and $w_k + r$, where w_k is the current value of connection weight and r is a randomly generated values between 0 and 1.

For node transfer functions, a mutation operator also creates a random index for each layer and replaces a function located at that position with a random transfer function from the predefined list. For evolving parameters in node transfer functions, the mutation operator changes a value of μ between -5.0 and 5.0. A value of σ changes between -5.0 and 5.0 for Gaussian function and between 1.0 and 5.0 for Sigmoid, Sigmoid Prime, Hyperbolic and Bipolar functions.

The generated population contains deep neural networks of different depth ranging from 1 to 5 hidden layers and it is not a subject to change during the evolutionary process. For an architecture, a mutation operator at first chooses a layer of the neural network, then it randomly chooses which operator to apply: *add* or *remove*. *Add* operator adds a new neuron to the previously chosen layer. *Remove* operators removes the last node from the layer. A probability of mutating an architecture of a deep neural network is 1%.

5 Results

For each benchmark problem 20 experiments were performed: 10 experiments for evolving a deep neural network with the sigmoid transfer function in each neuron and 10 experiments where node transfer functions were evolved for each neuron. Connection weights and architecture were simultaneously evolved during the NeuroEvolutionary process in all experiments. Each experiment contains 100 iterations.

The result of each experiment is a number, which represents the root-mean-square error (RMSE) associated with running a network on the test data set. RMSE is defined by means of the following formula:

$$C = \sqrt{\frac{\sum_{i=0}^{n}(y_i - t_i)^2}{n}}, \tag{1}$$

where:

n – a number of samples in the test data set;
C – RMSE;
y_i – the expected response of a neural network for the i_{th} sample from the test data set;
t_i – the actual response of a neural network for the i_{th} sample from the test data set.

Tables 3, 4 and 5 show results for each benchmark problem, where:

N – experiment number;
C_1 – RMSE when evolving connection weights and an architecture in a deep neural network;
C_2 – RMSE when evolving simultaneously connection weights and an architecture along with node transfer functions and its parameters in a deep neural network.

Table 6 shows an average RMSE associated with each evolutionary method.

Table 3. Results for Brodatz Textures.

N	1	2	3	4	5	6	7	8	9	10
C_1	0.69	0.68	0.68	0.7	0.68	0.71	0.69	0.69	0.69	0.67
C_2	0.63	0.66	0.65	0.65	0.6	0.66	0.64	0.63	0.64	0.63

Table 4. Results for handwritten digits.

N	1	2	3	4	5	6	7	8	9	10
C_1	0.66	0.68	0.68	0.67	0.68	0.71	0.66	0.67	0.69	0.67
C_2	0.65	0.66	0.66	0.65	0.65	0.64	0.66	0.65	0.65	0.65

Table 5. Results for COIL-100.

N	1	2	3	4	5	6	7	8	9	10
C_1	3.24	2.97	3.17	3.34	3.22	3.53	3.47	3.37	3.16	2.96
C_2	1.44	1.08	1.36	0.99	1.6	1.01	0.89	0.99	1.17	1.24

Table 6. Average RMSE for each evolutionary method and benchmark problem.

Cost\Benchmark	Brodatz Textures	Digits Recognition	COIL-100
$\langle C_1 \rangle$	0.688	0.677	3.243
$\langle C_2 \rangle$	0.639	0.652	1.177

6 Conclusions

In this paper we explored the effect of evolving a Node Transfer Function and its parameters, along with the evolution of connection weights and architecture in Deep Neural Networks for Pattern Recognition problems.

The results demonstrate that evolving both node transfer functions and the parameters for each node increases the performance of a deep neural network in pattern recognition problems. The performance boost is observed for all benchmark problems described in this paper. A **7%** boost was achieved for Brodatz Textures pattern recognition problem, **4%** for Digits Recognition and a dramatic **64%** boost for COIL-100. The improvement in performance for COIL-100 demonstrates that evolving a node transfer function can be used for an initial fine-tuning of a global minimum, which is an important discovery, because evolutionary algorithms are considered to be inefficient for this task.

This suggests the method of evolving node transfer functions and its parameters along with connection weights and architecture in deep neural networks should be considered to improve performance results for pattern recognition problems and should be included in research software for deep learning.

Further research can be conducted to extend a list of node transfer functions used for the described evolutionary process in deep neural networks.

References

1. Bengio, Y.: Learning deep architectures for AI. Found. Trends Mach. Learn. **2**, 1–127 (2009). Now Publishers
2. Pascanu, R., Montufar, G., Bengio, Y.: On the number of response regions of deep feedforward networks with piecewise linear activations. In: NIPS 2014, pp. 2924–2932 (2015)
3. Vincent, P., Larochelle, H., Bengio, Y., Manzagol, P.-A.: Extracting and composing robust features with denoising autoencoders. In: ICML, pp. 1096–1103 (2008)
4. Ranzato, M., Poultney, C., Chopra, S., LeCun, Y.: Efficient learning of sparse representations with an energy-based model. In: NIPS (2007)
5. Larochelle, H., Erhan, D., Courville, A., Bergstra, J., Bengio, Y.: An empirical evaluation of deep architectures on problems with many factors of variation. In: ICML, pp. 473–480 (2007)
6. Hochreiter, S., Bengio, Y., Franconi, P., Schmidhuber, J.: Gradient Flow in Recurrent Nets: The Difficulty of Learning Long-Term Dependencies. IEE Press, New York (2001)
7. Glorot, X., Bengio, Y.: Understanding the difficulty of training deep feedforward neural networks. In: AISTATS, pp. 249–256 (2010)
8. Sutskever, I., Martens, J., Dahl, G., Hilton, G.: On the importance of initialization and momentum in deep learning. In: ICML (3), vol. 28, pp. 1139–1147 (2013)
9. Kent, A., Williams, J.G. (eds.): Evolutionary Artificial Neural Networks. Encyclopedia of Computer Science and Technology, vol. 33, pp. 137–170. Marcel Dekker, New York (1995)
10. Yao, X.: Evolving artificial neural networks. Proc. IEEE **87**, 1423–1447 (2002)
11. David, O.E., Greental, I.: Genetic algorithms for evolving deep neural networks. In: GECCO, pp. 1451–1452 (2014)

12. Tirumala, S.S.: Implementation of evolutionary algorithms for deep architectures. In: AIC (2014)
13. Angeline, P.J., Saunders, G.M., Pollack, J.B.: An evolutionary algorithm that constructs recurrent neural networks. Neural Netw. **5**, 54–65 (1994)
14. Stanley, K.O., Miikkulainen, R.: Evolving neural networks through augmenting topologies. Evol. Comput. **10**, 99–127 (2002)
15. Mahsal, K.M., Masood, A.A., Khan, M., Miller, J.F.: Fastlearning neural networks using Cartesian genetic programming. Neurocomputing **121**, 274–289 (2013)
16. James, A.T., Miller, J.F.: NeuroEvolution: The Importance of Transfer Function Evolution (2013)
17. Vodianyk, D., Rokita, P.: Evolving node transfer functions in artificial neural networks for handwritten digits recognition. In: Chmielewski, L.J., Datta, A., Kozera, R., Wojciechowski, K. (eds.) ICCVG 2016. LNCS, vol. 9972, pp. 604–613. Springer, Cham (2016). doi:10.1007/978-3-319-46418-3_54
18. USC University of Southern California: Signal, Image Processing Institute, Ming Hsieh Department of Electrical Engineering. Textures, vol. 1. http://sipi.usc.edu/database/?volume=textures
19. The MNIST database of handwritten digits. http://yann.lecun.com/exdb/mnist/
20. Nene, S.A., Nayar, S.K., Murase, H.: Columbia Object Image Library (COIL-100). Technical report CUCS-006-96 (1996). http://www.cs.columbia.edu/CAVE/software/softlib/coil-100.php
21. Recommendation ITU-R BT.601-7 (2011)