A Comparison of Some Methods for Evolving Neural Networks

Marko Grönroos
Åbo Akademi University, Department of Computer Science
FIN-20520 Turku, Finland
Email: magi@utu.fi, Tel. +358 2 215 4072

Abstract

This paper presents an empirical comparison of four evolutionary encoding methods for finding suitable neural network topologies. We use five different learning problems for benchmarking the encoding methods. Three of the problems are artificial (Encoder and two function approximation problems), and two are real-world classification problems from the PROBEN1 benchmarking problem set. We train the network weights with a separate neural learning algorithm. Our evaluation criteria are classification accuracy and efficiency for using only the relevant variables.

1 Introduction

Artificial neural networks (ANNs) are a paradigm for implementing complex nonlinear mappings using simple elementary units that are connected together with weighted connections. Although many quite efficient algorithms exist for the determination of the connection weights, no definitive methods exist for finding connection architectures most suitable for particular problems. Evolutionary design of the architectures is one such approach.

Many evolutionary methods for such design have been developed during the recent years and the results have been encouraging. However, we can observe that most of the methods have been tested in relation to only a single problem, which has often been a simple toy problem or an artificial life problem. Therefore, we set as our main goal of this study to compare the capabilities of some of these methods by applying them to various (more or less) difficult function approximation and pattern recognition problems. Three of the problems used in this study are artificial problems for which we can easily deduce very suitable neural architectures. They are followed by two real-world classification problems from the PROBEN1 problem set [10] that are well known for benchmarking purposes in the neural learning field.

We compare empirically four methods for the evolutionary design of ANN architectures. We concentrate on encoding methods that can be used solely for the design of the network connectivity, as we do the actual training of the network weights with a conventional gradient descent neural learning method. The evolutionary algorithm (EA) uses the error of such a trained network as a fitness value to guide the evolution of a population of neural architectures.

We use two criteria for the evaluation of the performance of the methods: first is the classification accuracy of the best neural network topology found by an evolutionary search. The second criterion is the selection of relevant input variables, which is most meaningful with the artificial problems where we already know the correct answer. We can expect the more direct network encoding methods to have stronger control over the individual neurons, and therefore perform better in this task.

We first describe the encoding methods, then the problem datasets, the algorithms and parameters we used in our experiments, the results from the experiments, a short discussion about the results, and finally the conclusions. These subjects are elaborated in [4].

2 Encoding methods

The central task in evolving ANNs is finding a genetic representation, or encoding, for an ANN. It dictates how the search landscape is structured, and how scalable the method is. There are currently maybe dozens of different encoding methods. Some use binary-

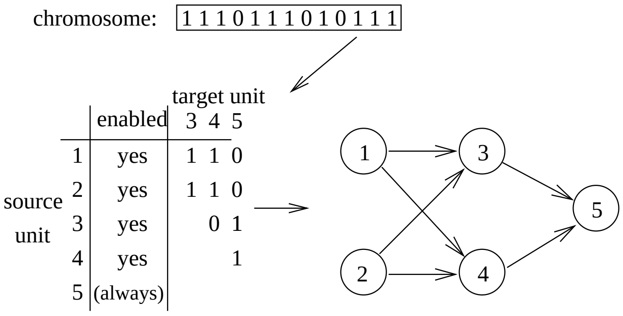

Figure 1: Direct Encoding

valued vector representations, while others use tree-structured genomes, etc.

The first two methods studied in this work, namely the direct encoding and the graph generation grammar, were selected mostly because they are so well-known and easy to implement. The cell space encoding was selected because it had a temptating space-coordinate encoding that was hoped to yield networks that are modular and visually simple. The generative cell space encoding was selected because it is a derivative of the previous one and uses a temptating generative growth process.

The biggest difference between the original methods and the implementations evaluated in this study is that two of the original methods did not use a separate neural learning algorithm, but the connection weights were also adapted by the EA.

2.1 Direct encoding

We use a direct encoding method derived from the one proposed by Miller, Todd, and Hedge [7]. Neurons are indexed $1...N$. For each potential feedforward connection from neuron i to non-input neuron j (when $i < j$), we have one binary gene that indicates whether (1) or not (0) that connection exists. We encode also the existence of each neuron with a binary gene, as we want to be able to control the selection of the input units more efficiently. The output units are hard-coded to always exist. An example of the decoding process is shown in Figure 1.

The connectivity of the initial networks is controlled with probability of connection p_c from a neuron to another. We use $p_c = 0.5$ in our experiments.

2.2 Towards fractal encodings

To build complex neural networks, researchers have (again) looked at how it has been done in biological systems. The mapping from genotype to phenotype is definitely not direct in biological organisms. We are not built according to direct "blueprints" but according to indirect "recipes" (using the expressions of Dawkins [2]). The "cooking process" is called the *ontogenesis* of an organism; the long and complex process or splitting and specialization of cells according to the rules encoded in the genome, with the adult individual as the result.

Many of the current evolutionary neural network methodologies have been inspired by the *Lindenmayer-Systems* or L-Systems. The formalism and an application of this idea was introduced by Lindenmayer [6], originally as a *grammatic* approach for modeling plant morphogenesis. The grammars of an L-system consist of a set of *production rules* that are used to generate a morphological description string. The process of applying the rules is called *string-rewriting.* We start the string-rewriting with an initial string "S". Each left-hand side of a rule found in the current string is rewritten (replaced) by the right-hand side. For example, if we apply the rule

$$S \to aSb \tag{1}$$

to the initial string "S", we get the string "aSb". When we apply the rule again, we get "aaSbb". The rules are usually applied for some predefined number of iterations. Each string which has been generated in such manner must have a sound interpretation to be useful. Lindenmayer interpreted the symbols as commands to guide a "turtle" for drawing pictures of plants.

Another famous version of rewriting is the one by Dawkins [2], who used nine cell division parameters to evolve fractal graphical "organisms" called *morphs,* that are similar to the plants generated with L-systems.

Original purpose for the L-systems was the creation of *forms* of plants, but creating something that *looks* like a plant does not mean that it was constructed in the same way the plant actually grows. Biological trees are grown by dividing cells, not with turtles, and they definitely do not contain production rules, but just DNA and proteins. The computational mechanism that controls the biological growth process is called *genetic regulation.* String-rewriting systems have been used as idealizations of genetic regulation and cell differentation in many studies dealing with evolutionary ANNs in the last few years. Some approaches have used *genetic regulation networks* (which can be genetically encoded recurrent ANNs) for this idealization instead of grammars.

2.3 Graph generation grammar

The *graph generation grammar* (GGG) developed by Kitano [5] is an early grammar encoding method based on context-free and deterministic L-systems. The grammar contains production rules of the form

$$A \rightarrow \begin{bmatrix} B & C \\ D & E \end{bmatrix}$$

where the left-hand-side (LHS) of the production is a symbol and the right-hand-side (RHS) is a 2×2 matrix of symbols from the alphabet $\Sigma = N \cup P \cup T$, where $N = \{A, B, ..., Z\}$ are *nonterminal*, $P = \{a, b, ..., p\}$ are *preterminal*, and $T = \{0, 1\}$are *terminal* symbols. The grammar is constructed from two parts: the *variable part*, and the *constant part*. The variable part is the one encoded in the genome and its LHS symbols are nonterminals and RHS symbols are in $N \cup P$. An example genome is illustrated in Figure 2a. The start symbol S is guaranteed to exist as the LHS of the first rule in the genome. The constant part contains 16 rules which have the preterminals $\{a, b, ..., p\}$ as their LHS and the corresponding 16 possible 2×2 matrices of *terminal* symbols $T = \{0, 1\}$ on the RHS matrix. Figure 2b shows an example of such set of rules. We call the symbols in T as "terminal", although they are still rewritten with constant rules

$$0 \rightarrow \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} \quad \text{and} \quad 1 \rightarrow \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}.$$

It is common during the rewriting process that there exists no rule in the grammar for a certain nonterminal in the matrix. Those nonterminals are interpreted as "dead", and are rewritten using the same rule as for zeros (as is done for the letter M in Figure 2c). If there are two rules with the same LHS in the genome, the first one is used. The initial symbol is rewritten for a predetermined number of iterations, as done in the example in Figure 2c. In the final iteration step, all nonterminals still present in the matrix are replaced with zeros, as is done for the letter "A" in our example.

The diagonal bit of the final matrix indicates the presense or absense of a specific neuron, as in the direct encoding. To make the network topology feedforward, any feedback connections (lower left part of the connection matrix) are removed from the connection matrix. Figure 2d illustrates the final network generated from the example genotype.

In this study, we encode 64 variable production rules in the genome of each individual, with 26 nonterminals $\{A, B, ..., Z\}$. As with the direct encoding, the output units are forced to be enabled.

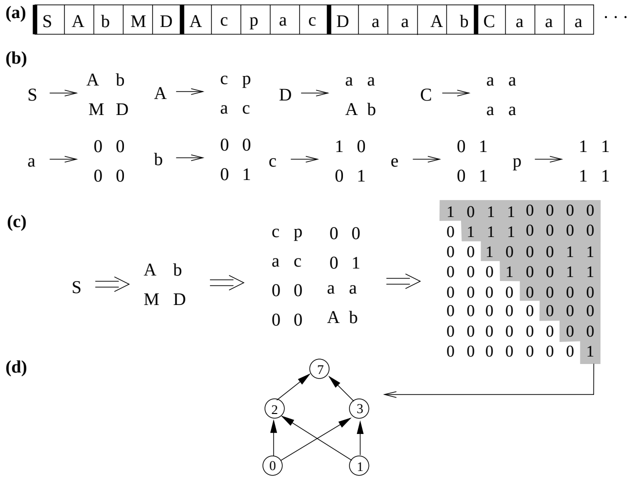

Figure 2: GGG Encoding

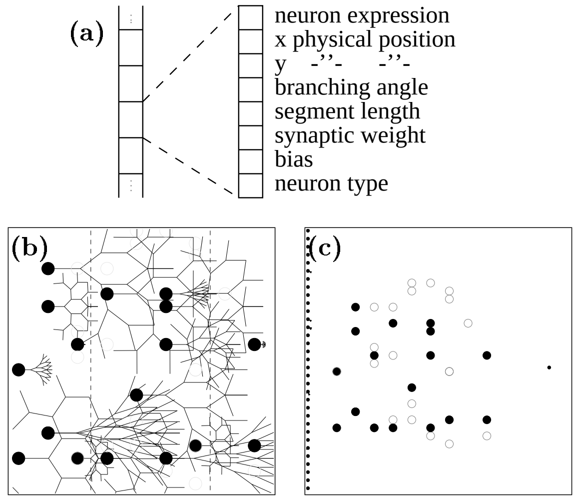

Figure 3: Construction of a Nolfi and Parisi Network

2.4 Cell space encoding

In the encoding method by Nolfi and Parisi [8, 9], the neurons are encoded with coordinates in a two-dimensional space. The mapping from genes to neurons is direct in a sense, but the connections are grown in a special manner. The phases of the decoding process are illustrated in Figure 3. Those neurons which are located in the left part of the space are considered as input units and those in the right part are considered as output units.

Our implementation of the decoding process of the *cell space* (CS) *encoding* proceeds as follows. The neuron index value i of an input or output neuron u_i is determined according to a *type* gene. For input units the index is $i = \mathit{type} \, \mathbf{mod} \, N_\iota$ and for output units $i = N - N_o + \mathit{type} \, \mathbf{mod} \, N_o$, where N_ι is the number of inputs, and N_o the number of outputs. Using this indexing, several neurons can clearly have the same index. Therefore, the connection of input and output units is handled in a special manner for this encoding method. We add "real" input and output layers where each "real" input unit is connected to every encoded input unit of the corresponding "type", and likewise for the output units. The index values of hidden neurons are determined according to their x-coordinate. If two neurons have the same x-coordinate, their index value is determined by their order in the genome.

The connections are determined by letting "axon trees" grow forward from neurons. This is illustrated in Figure 3b. The axon trees are basicly L-system fractals, although the exact generation method probably doesn't matter much. The segment length and the branching angle of the fractal are individually encoded for each neuron. A connection is made when an axon tip touches another neuron, i.e. comes within distance r_c from the axon tip to the center of a target neuron. Nolfi and Parisi apparently used $r_c = 0.5$. We use $r_c = 2$ in this study. We discard feedback connections according to the index values of source and target neurons of a connection. An almost final network is shown in Figure 3c; some useless neurons can be removed.

Nolfi and Parisi did not use a separate training algorithm to determine the weights and biases, as they were also encoded in the genome for each neuron. They used the encoding method for the study of artificial life, concerning mostly the search of "food" and "water" sources by an autonomous artificial animal.

2.5 Generative cell space encoding

The fourth method was selected because it is a derivative of the previous one, and employs a fractal growth process similar to the method by Kitano.

The above method by Nolfi and Parisi was further developed by Cangelosi, Parisi and Nolfi [1]. They added cell division and migration rules to *grow* the neuron population instead of encoding each neuron directly. Thus we call their *generative cell space* (GCS) *encoding*.

The ontogenesis is started with one "egg" cell with a specific start cell type. The cell is split (rewritten) into two daughter cells with the modifications to attributes given by the applied rule. The daughter cells are positioned in the 8 possible locations in the grid around the mother cell. The division is repeated for a predefined number of iterations, after which the cells "maturate" as neurons. The neurons become input, hidden, or output neurons according to the same spatial rule as

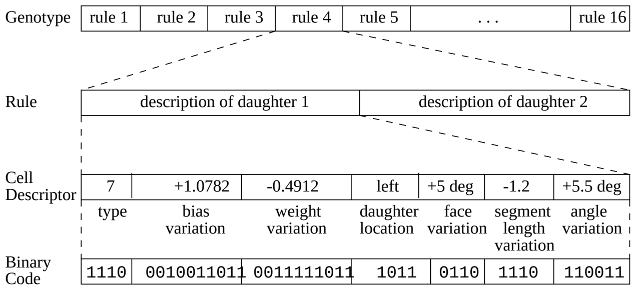

Figure 4: Genome in GCS Encoding

in regular cell space encoding and their exact identity is determined by the final cell type.

Cangelosi et al. used the method to evolve control networks for artificial lifeforms, and did not use a separate neural learning algorithm. In this study, we do not use the *face*, *weight*, and *bias* genes used by Cangelosi et al.

2.6 Discussion

Siddiqi and Lucas [12] have also made a comparison between the direct and GGG encodings, using the same encoder problem as Kitano used in his comparison [5], and we use here. They noted that the comparison made by Kitano was unfair for the direct encoding because Kitano used connection propability $p_c = 0.3$ for the direct encoding, which is about the same as the GGG would give. They found that the direct encoding performed better with higher p_c. We can, however, expect that this optimal connectivity depends on the problem, so the better p_c found by Siddiqi and Lucas might not hold for other problems (and hence we use $p_c = 0.5$); it is a very good starting-point in the encoder problem that all the inputs are enabled and connected to all the outputs, which would be achieved with $p_c = 1.0$. That would directly give a working solution that could then be pruned to learn a bit faster. The notion of Siddiqi and Lucas does, however, bring forth one point: it is not as easy to adjust the connection probability for the grammar encoding as it is for the direct encoding.

The histogram in Figure 5a depicts the distribution of the p_c for initial genomes of the direct and GGG encodings with the `encoder` problem (see below). The histogram has been calculated from 1000 random individuals. The GGG has a very wide distribution, so the probability of getting a highly connected network in the initial population is higher. Obviously, it would be possible to adjust the distribution of initial connectivity of the direct encoding by a special initialization procedure for the binary genes. It is not possible to calculate p_c for the other two encoding methods, but we can examine the distribution of number of connections in the networks, shown in Figure 5b. It seems that, regardless of the p_c, GGG creates smaller networks than direct encoding in the initial population. The CS and GCS encodings produce a very high number of fatal networks.

3 Datasets

We use three artificial problems to benchmark the input selection ability. First is the classic 8-x-8 encoder problem. The x in the encoder name refers to the fact that the number of hidden neurons is determined by the EA, as is the network topology generally. We use this problem because Kitano used it in his study [5]. Clearly, no hidden units are required at all, but the connections can be made directly from the input to the output layer. This makes the encoder problem rather mundane.

The `additive` data has been generated using a function that has a nonlinear additive dependence on the first two variables, a linear dependence on the next three and is independent of the last five (pure noise) variables. We use a function

$$\begin{aligned} f(x) = \; & 0.1e^{4x_1} + \frac{4}{1+e^{-20(x_2-1/2)}} \\ & +3x_3 + 2x_4 + x_5 + 0 \cdot \textstyle\sum_{i=6}^{10} x_i \end{aligned} \quad (2)$$

presented by Friedman [3]. The second term is clearly the sigmoid function, so it can be modeled to arbitrary accuracy with just one neuron. The training set used in our tests contains 50 ten-dimensional covariate vectors generated in the unit hypercube. We assign the corresponding response values according to

$$y_i = f(x_i) + N(0,1) \quad (3)$$

where $f(x)$ is given by (2) and $N(0,1)$ is a normally distributed random variable with mean 0 and standard deviation 1. Linear output units must be used for this and the following problem.

The `interaction` dataset was generated in a manner simular to the `additive` data, using a function that has a two-variable nonlinear depence, a one-variable nonlinear dependence, two linear dependences and is independent of the last five (pure noise) variables:

$$\begin{aligned} f(x) = \; & 10\sin(\pi x_1 x_2) + 20(x_3 - 1/2)^2 \\ & +10x_4 + 5x_5 + 0 \cdot \textstyle\sum_{i=6}^{10} x_i . \end{aligned} \quad (4)$$

The training set for this problem contains 200 patterns.

Table 1: Dataset characteristics

Dataset	**#in**	**#out**	**Number of patterns**			
			Train	E.s.	Valid.	Test
`encoder`	8	8	8	-	8	8
`additive`	10	1	37	13	50	1000
`interaction`	10	1	150	50	200	1000
`glass1a`	9	6	80	27	54	53
`heart1a`	35	2	345	115	230	230

We use two real-world classification problems from the PROBEN1 database: `glass` and `heart`. These problems are well known in machine learning benchmarking. We use the "`1a`" partitioning and permutation for the PROBEN1 datasets, i.e. `glass1a` and `heart1a`.

The datasets (except the `encoder`) are divided into four parts, of which one is used for neural learning, one for early stopping in neural learning, one for evaluation of fitness, and one for final testing. The set used for the evaluation of fitness is used for early stopping in the final testing. The number of patterns in each set is shown in Table 1, which also gives the number of input and output units for each problem.

4 Experiments

We used an evolutionary algorithm to evolve network topologies for every dataset with all four encoding methods. One to four runs were done for each method-problem pair; only one solution was evolved with the CS and GCS encodings for the classification problems.

Each population consisting of 50 individuals was evolved for 100 generations. The next generation was formed by selecting and mating from the 8 fittest individuals with uniform probability. The best individual was always preserved unchanged to the next generation (elitism). Recombination frequency was 0.5. We used self-adaptation of mutation rates in probability range $[0.0001, 0.01)$ for both binary and multi-valued genes. The latter gene type was required by the GGG encoding.

The number of potential neurons, was a power of two for all problems; 32 for all but the `heart` problem, for which we used 64 neurons. The number was same for all encoding methods, except for the runs 3 and 4 with the CS encoding, where the number of encoded neurons was doubled. It was reasonably higher than the total number of inputs and outputs to leave room for hidden units. The number of neuron types for the CS and GCS encodings was 8 for the `encoder` problem,

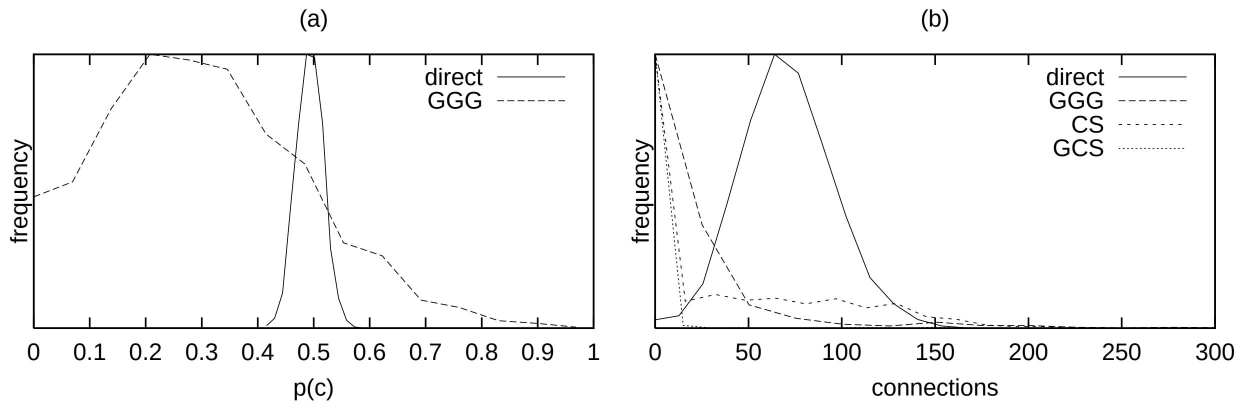

Figure 5: Frequency Distributions of Initial Connectivities

64 for the `heart` problem, and 16 for other problems.

We trained the network weights using the resilient backpropagation (RProp) algorithm, with learning parameters $\Delta_{min} = 1e^{-6}$, $\Delta_{max} = 50.0$, $\eta^- = 0.5$, $\eta^+ = 1.2$, and weight decay of 0.9999. We initialized the weights to range $[-0.5, 0.5]$ with uniform distribution. We used GL_2 early stopping [11] during evolution and GL_5 in final testing, except for the the encoder problem which does not need early stopping. The maximum number of training cycles was 3000, except for the `encoder` problem, for which we used a relatively small value of 50. We did this because we wanted to avoid the RProp learning the problems too well, i.e. reaching the calculation precision limit and thus leaving no decent gradient for the EA to evolve on.

The best network topology of the final population of each evolution run was trained and tested for 30 times with different connection weight initializations. After training, the networks were tested with final test sets. The classification results are given in Table 2. The values for the direct and GGG encodings are averaged from the four fittest networks found by the evolution runs. Results obtained from test runs with the best topologies found by Pretchelt are given for comparison ("S" denotes shortcuts between non-adjacent layers and "NS" no shortcuts). The best performing encoding method is emphasized. The MSEs (mean squared errors) are multiplied by 100. Standard error for the standard deviation σ with 30 tests is $\sigma_s = \sigma/\sqrt{30}$ ($\sqrt{30} \approx 5.5$). We can use this statistic to say, for example, that about 95% of initializations with `glass` problem and GGG encoding give classification error in range $33.46 \pm 2 \cdot 5.09/\sqrt{30} = 31.60...35.32$. However, as demonstrated below, the distribution may not be normal. The topology found by Prechelt gave better classification results (shown in italics) for both problems, but those differences are not at all significant, and if measured by the MSEs, the evolved networks performed better for the `glass` problem.

The input selections are shown in Table 3. Summary of the input selection by different methods for different problems. The use of a certain input is denoted with $\bullet$, and leaving out with $\circ$. The inputs are ordered in increasing index order from left to right. All the inputs should have been enabled for the encoder problem, as done always by the direct and GGG encodings, but GCS failed quite badly in that task. For the `additive` and `interaction` problems, only the first five inputs should have been enabled. GGG performed quite bad for those problems, typically selecting the first 8 inputs. CS performed clearly best for these problems, and GCS left again many meaningful inputs unused.

5 Discussion

We can observe in Table 2 that solutions that are worse on average usually have larger deviation. This situation seems to be caused by too small networks which are more difficult to train, and the training often ends in a very bad solution. The best solutions of such small topologies can, however, be better than with bigger networks.

Table 4 presents a comparison of three hand-picked topologies with 1000 training-testing-runs with different random network initializations. The first one, 35-8-2 (35 input neurons, 8 hidden neurons and 2 outputs), is the same used by Prechelt in his sample tests (see the first row). The amount of noise in the fitness was estimated by training and testing the `heart` data 1000 times with three fixed neural topologies that were somewhat different from each other. The distri-

Table 2: Classification Results

Problem	Method	Runs	Training set clsf. err.% mean	Training set clsf. err.% σ	Training set MSE mean	Training set MSE σ	Validation set clsf. err.% mean	Validation set clsf. err.% σ	Validation set MSE mean	Validation set MSE σ	Test set clsf. err.% mean	Test set clsf. err.% σ	Test set MSE mean	Test set MSE σ
glass1a	direct	4	34.17	8.74	8.75	2.29	40.70	7.27	10.38	2.20	42.00	6.74	11.31	2.58
	GGG	4	29.78	6.58	7.12	1.04	34.78	3.06	9.18	0.27	**33.46**	5.09	9.44	0.49
	CS	1	37.85	14.63	10.93	4.66	43.40	12.24	12.10	4.56	49.69	9.37	14.71	5.90
	GCS	1	62.68	14.03	16.28	7.51	59.81	15.90	15.62	6.80	63.46	14.89	16.08	6.37
	prechelt S		33.36	6.25	7.92	0.81	36.11	2.65	9.54	0.30	*33.14*	5.20	10.16	0.64
	prechelt NS		32.93	5.56	7.82	1.03	37.10	3.41	9.41	0.74	34.47	6.83	9.86	1.14
heart1a	direct	4	15.89	2.29	12.31	2.15	17.97	2.22	13.38	1.91	21.14	1.88	15.62	1.89
	GGG	4	14.30	1.58	11.19	1.45	18.25	1.67	13.57	1.26	**20.53**	1.41	14.75	1.24
	CS	1	22.70	9.64	18.48	10.83	24.52	9.40	19.63	10.72	26.84	7.10	21.78	9.28
	GCS	1	32.41	12.22	24.72	9.97	33.80	12.63	25.52	9.92	34.16	10.06	26.10	9.35
	prechelt S		12.27	0.94	10.32	0.53	17.65	1.25	13.53	0.37	*19.41*	1.18	14.61	0.39
	prechelt NS		12.62	0.92	10.13	0.56	18.80	0.97	13.36	0.37	19.93	0.98	14.41	0.35

Table 3: Input Selection Results

Method	Run	Problem: Encoder	Problem: Additive	Problem: Interaction
direct	1	●●●●●●●●	●●●●●○○○○○	●●●●●●○○○●
	2	●●●●●●●●	●●●●●○○○○●	●●●●●●○○●○
	3	●●●●●●●●	●●●●●○○○○●	●●●●●○○○●○
	4	●●●●●●●●	●●●●○○○○○○	●●●●●○○○○●
GGG	1	●●●●●●●●	●●●●●●●●○○	●●●●●●●●○○
	2	●●●●●●●●	●●●●●●●●○○	●●●●●●●●●●
	3	●●●●●●●●	●●●●●●●●○○	●●●●●●●●○○
	4	●●●●●●●●	●●●●●●●●○○	●●●●●●●●○○
CS	1	●○●●●●○○	●●●●●○○○●○	●●●●●○○○○○
	2	●●●○○●●●	●●○●●○○○○○	●●●●●○○○○○
	3	●●●●●●●●	●●●●●○●○○○	●●●●●○○○○○
	4	●●●●●●●●	●●○●○○○○○○	●●●●●○○○○○
GCS	1	●●●○○○○○	●●○●○○○○○○	○●○●●○○○●○
	2	○●●○○○○●	●●○●●○○○○○	●●○●●○○○○○
	3	●○○●●●○○	●●○●○○○○○○	●●○●●○○○○○
	4	○○●○○●○●	●●○●○○○○○○	●●●●●○○○○○

Table 4: Effects of Network Size

Method	Test set clsf. err.% mean	Test set clsf. err.% σ	Test set MSE mean	Test set MSE σ
Prechelt	19.89	2.27	14.33	1.26
35-8-2 S	19.53	1.29	14.58	0.47
35-4-2 S	19.74	1.25	14.56	0.42
35-2-2	21.99	5.07	19.73	10.41

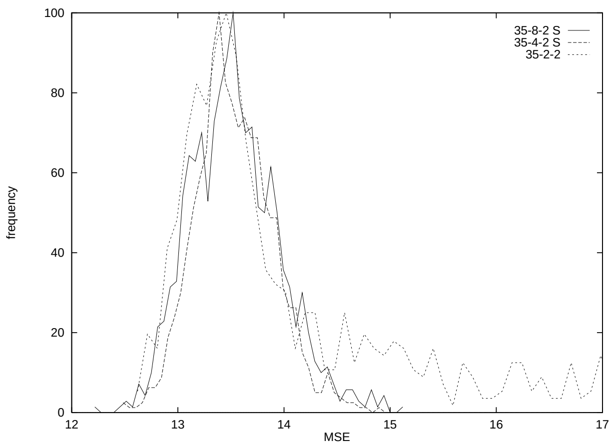

Figure 6: Effects of Network Size

butions are visualized in histograms in Figure 6. The histograms were constructed according to the range of each test case separately, so their scaling varies. The resolution (number of slots) of the histogram was 1000, 100 and 50 for the sets 35-8-2S, 35-4-2S and 35-2-2, respectively (S denotes shortcut connections between non-adjacent layers). The frequency was scaled so that the maximum frequency for each histogram is 100. The mean MSE for the 35-2-2 network is actually outside the range of the graph.

This demonstrates just how much the network complexity dictates the learning behaviour. We would like to prefer smaller networks, as they can be expected to generalize better, but evolving such smaller networks is not simple, because of the erratic learning behaviour shown above. Many studies have used complexity terms in fitness values, with good results. It might also be that the selection parameters of the EA affect this behaviour, especially elitism.

6 Conclusions

We evolved artificial neural network topologies using four different genetic encoding methods: the direct encoding by Miller et al., graph generation grammar by Kitano, cell space encoding by Nolfi and Parisi, and generative cell space encoding by Cangelosi et al. The network weights were trained using the RProp algorithm. The fitness of the network topologies was calculated using the validation error with one of five problems during the evolutionary search. The best network topologies found by the evolutionary search runs were compared by their classification accuracy with the real-world problems, and by their ability to select the relevant inputs with the artificial problems.

The network topologies evolved with the graph grammar encoding gave generally the best classification results, but the direct encoding was not far behind. The cell space encoding performed worse than the two methods above, and the generative cell space encoding was the weakest. The classification performance of the two best methods was about the same as with network topologies found by a systematic search for the PROBEN1 datasets [10].

The direct encoding was perhaps the best for selecting the correct inputs, although the cell space encoding was not far behind. The graph grammar encoding performed clearly worst in this task, because it had great difficulties in generating small details in the network description matrix it used. The generative cell space encoding performed again worst.

Acknowledgements

This study was done within the Countess (Computational Intelligence for Business) project, which is a long-term project done in collaboration with the Department of Information Processing of the Turku University of Economy. We wish to thank Kaisa Sere and Barbro Back for the possibility to work in the project, Olli Nevalainen for many corrections, and Christian Lehtinen for preparing the `additive` and `interaction` datasets.

References

[1] A. Cangelosi, D. Parisi, and S. Nolfi. Cell division and migration in a 'genotype' for neural networks. *Network: Computation in Neural Systems*, 5:497–515, 1993.

[2] R. Dawkins. *The Blind Watchmaker*. New York: Norton and Cooper, 1986.

[3] J.H. Friedman. Multivariate adaptive regression splines. *The Annals of Statistics*, 19(1):1–141, 1991.

[4] M. A. Grönroos. Evolutionary design of neural networks. Master's thesis, Computer Science, Department of Mathematical Sciences, University of Turku, Finland, 1998. Available from http://www.utu.fi/~magi/opinnot/gradu/.

[5] H. Kitano. Designing neural network using genetic algorithm with graph generation system. *Complex Systems*, 4:461–476, 1990.

[6] A. Lindenmayer and G. Rozenberg. *Automata, Languages, Development*. Amsterdam, North-Holland, 1976.

[7] G.F. Miller, P.M. Todd, and S.U. Hegde. Designing neural networks using genetic algorithms. In J.D. Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*, pages 379–384. Morgan Kaufmann, 1989.

[8] S. Nolfi and D. Parisi. Growing neural networks. Technical report, Institute of Psychology, CNR, Rome, 1992.

[9] S. Nolfi and D. Parisi. Genotypes for neural networks. In M. A. Arbib, editor, *The Handbook of Brain Theory and Neural Networks*. Cambridge, MA: MIT Press, a Bradford book, 1994.

[10] L. Prechelt. PROBEN1 - a set of neural network benchmark problems and benchmarking rules. Technical Report 21/94, Fakultät für Informatik, Universität Karlsruhe, 76128 Karlsruhe, Germany, 1994.

[11] L. Prechelt. Automatic early stopping using cross validation. *Neural Networks*, 11, 1998.

[12] A. A. Siddiqi and S. M. Lucas. A comparison of matrix rewriting versus direct encoding for evolving neural networks. In *Proceedings of the 1998 IEEE International Conference on Evolutionary Computation (ICEC'98)*, pages 392–397. IEEE, 1998.