Parallel Cascade Correlation: A GPU Implementation

Firas Safadi, Raphaël Fonteneau and Damien Ernst

Abstract

This paper presents a new implementation of the cascade correlation architecture which leverages the parallel computing capabilities of GPUs. It shows that by combining the inherent parallelization potential of evolutionary algorithms with the caching properties of the cascade correlation architecture, the algorithm can be simplified to a few easily parallelizable operations. It also comes with an open-source CUDA C++ implementation of the resulting genetic cascade correlation algorithm.

1 Introduction

Today, parallel computing is highly accessible and widespread thanks to cheap GPUs with thousands of cores and well-documented APIs. New algorithms are therefore being designed with parallelization in mind. Existing ones however need to be rethought and adapted to parallel architectures. In this article, the genetic cascade correlation algorithm is reexamined and implemented for NVIDIA's parallel computing architecture, CUDA.

Cascade correlation is a supervised learning algorithm which adaptively builds a neural network during the training phase and presents several advantages compared to other algorithms, such as learning very quickly [8]. It does however suffer from overfitting issues [10, 18].

In the genetic cascade correlation algorithm, the standard Quickprop optimization algorithm used in the original cascade correlation algorithm is replaced with a genetic algorithm, which presents several benefits such as reducing the possibility of convergence to local optimums instead of global ones and support for applications in which the gradient of the optimization function cannot be computed [17].

This work focuses on the parallelization advantages of using genetic algorithms in the cascade correlation learning architecture. It shows how the resulting algorithm can be simplified to a few basic operations which are easy to

parallelize or for which optimized parallel solutions already exist. The CUDA C++ implementation of the algorithm can be downloaded online [2].

2 Cascade Correlation

The cascade correlation architecture is a type of neural network characterized by a number of features. In this section, it is described in the context of regression where the training dataset is composed of $n \in \mathbb{N}^*$ input-output pairs, or samples, where the input is a scalar vector of size $a \in \mathbb{N}^*$ and the output is a scalar:

$$\mathcal{D} = \{(\boldsymbol{x}_1, y_1), ..., (\boldsymbol{x}_n, y_n)\},\ \forall i \in \{1, ..., n\}, \boldsymbol{x}_i \in \mathbb{R}^a, y_i \in \mathbb{R}$$

2.1 Network Topology

The network is automatically built by adding hidden layers during the training phase. Each hidden layer contains one neuron which outputs a signal computed using the inputs of the hidden layer and which serves as an additional input for all subsequent layers. In addition, network inputs are directly connected to all layers. The number of inputs of a hidden layer is thus equal to that of the previous layer plus one. Any layer in the network can be described using a vector of scalar weights. The network itself can be described as a vector of layers. Initially, the network starts with only an output layer connected to the network inputs as well as a bias signal. The network starts as:

$$\boldsymbol{N}_0 = (\boldsymbol{o}_0),\ \boldsymbol{o}_0 \in \mathbb{R}^{a+1}$$

After l hidden layers have been added, the network becomes:

$$\boldsymbol{N}_l = (\boldsymbol{h}_1, ..., \boldsymbol{h}_l, \boldsymbol{o}_l),\ \boldsymbol{h}_1 \in \mathbb{R}^{a+1}, ..., \boldsymbol{h}_l \in \mathbb{R}^{a+l}, \boldsymbol{o}_l \in \mathbb{R}^{a+l+1}$$

Thus we have that:

$$\begin{aligned}
&\boldsymbol{h}_l \in \mathbb{R}^{a+l}\ \forall l \in \mathbb{N}^* \\
&\boldsymbol{o}_l \in \mathbb{R}^{a+l+1}\ \forall l \in \mathbb{N} \\
&\boldsymbol{N}_l \in \begin{cases} \mathbb{R}^{a+1} & \text{if } l = 0 \\ \mathbb{R}^{a+1} \times ... \times \mathbb{R}^{a+l+1} & \text{if } l \geq 1 \end{cases}
\end{aligned}$$

The number of weights in the output layer increases as the network grows. The final size of the network can be determined using different criteria, such as reaching a residual error goal or a maximum number of hidden layers.

2.2 Network Construction

The algorithm for adding a hidden layer to the network works as follows. First, the weights of the output layer are trained to minimize $E_l,\ l \in \mathbb{N}$, the total

error of the network with l hidden layers over the training dataset. Then, as long as some stopping criteria is not met, the following operations are repeated. The output layer is temporarily disconnected from the network and replaced with a new hidden layer. The weights of the new hidden layer are then trained to maximize the absolute value of the covariance between its output signal and the residual error in the network over the training dataset. More precisely, the maximized value $S_l,\ l \in \mathbb{N}^*$ is defined for a hidden layer as

$$S_l = \left| \sum_{i=1}^{n} (v_{i_l} - \bar{v}_l)(e_{i_l} - \bar{e}_l) \right|$$

where v_{i_l} is the output value of the lth hidden layer for the ith sample in the training dataset, $\bar{v}_l$ is the mean output value of the lth hidden layer over all samples, e_{i_l} is the observed network error for the ith sample before adding the lth hidden layer and $\bar{e}_l$ is the mean observed network error over all samples before adding the lth hidden layer.

Once trained, the hidden layer is permanently added to the network and its weights are frozen. The output layer is reconnected with an additional weight for the new connection and is retrained. Because of the correlation between the output signal of the new hidden layer and the residual error in the network, the latter should be reduced further when the weights of the output layer are retrained. A pseudocode for this algorithm is presented below.

Algorithm 1 Network construction algorithm

$l \leftarrow 0$
$\boldsymbol{H} \leftarrow ()$
$\boldsymbol{o} \leftarrow rand(a+1)$
$train(\boldsymbol{o})$
while not *stop* **do**
  $l \leftarrow l+1$
  $\boldsymbol{h}_l = rand(a+l)$
  $trainh(\boldsymbol{h}_l)$
  $\boldsymbol{H} \leftarrow \boldsymbol{H} \parallel \boldsymbol{h}_l$
  $\boldsymbol{o} \leftarrow \boldsymbol{o} \parallel rand()$
  $train(\boldsymbol{o})$
end while
$\boldsymbol{N} = \boldsymbol{H} \parallel \boldsymbol{o}$

In this code, the *rand* function returns a random vector of specified dimension or a random scalar if no dimension is specified. The *train* procedure tunes the weights of the output layer $\boldsymbol{o}_l$ so as to minimize E_l, whereas the *trainh* procedure tunes the weights of the new hidden layer $\boldsymbol{h}_l$ to maximize S_l. Note that because the weights of the hidden layers are frozen, training the network is reduced to training the output layer.

3 Genetic Algorithms

Genetic algorithms are an optimization technique inspired by biological evolution. A genetic algorithm involves maintaining a population of individuals each representing a solution to the optimization problem. The performance or quality of individuals can be evaluated using a fitness function and is improved by repeatedly evolving them using genetic operators such as selection, combination and mutation. In this section, the technique is described in the context of optimizing weights in a neural network. Individuals are therefore represented by scalar vectors.

3.1 Weight optimization

Optimizing a weight vector of dimension b using genetic algorithms works as follows. First, a population $\boldsymbol{P}$ is created by randomly generating many individuals. Let p be the number of individuals in the population. $\boldsymbol{P}$ is a $b \times p$ matrix where each column is an individual in the population:

$$\boldsymbol{P} = (\boldsymbol{d}_1, ..., \boldsymbol{d}_p),\ \boldsymbol{d}_1, ..., \boldsymbol{d}_p \in \mathbb{R}^b$$

The population is evaluated using a fitness function $f : \mathbb{R}^{b \times p} \to \mathbb{R}^p$ specific to the optimization problem which measures the quality q of the solution represented by an individual by assigning to it a positive scalar called fitness which grows larger as the quality improves:

$$f(\boldsymbol{P}) = \boldsymbol{q} = (q_1, ..., q_p),\ q_j \geq 0\ \forall j \in \{1, ..., p\}$$

The population is then evolved using genetic operators g times, where g is the number of epochs in an evolution cycle. The fitness vector $\boldsymbol{q}$ is used to eliminate weak individuals and filter out weak genetic material from one generation to the next, keeping strong genetic material and using it to create stronger solutions. It is computed at the end of each epoch. After g epochs, the evolution cycle is complete and the individual with the highest fitness $\boldsymbol{d}_s \mid q_s \geq q_j\ \forall j \in \{1, ..., p\}$ is used as the optimal solution. A pseudocode for this algorithm is shown below.

Algorithm 2 Weight vector optimization

```
P ← rand(b, p)
q ← f(P)
for i ← 1 to g do
    P ← evolve(P, q)
    q ← f(P)
end for
s ← imax(q)
```

Here, the *rand* function returns a random matrix of the specified dimensions. Given a population and fitness vector, the *evolve* function generates a new

population using genetic operators. The *imax* function finds the index in a vector of the component with the highest value.

4 Cascade Correlation using Genetic Algorithms

The merits of using genetic algorithms in the cascade correlation architecture are discussed in [17]. In addition to the theoretical advantages, the genetic cascade correlation algorithm is very well suited for parallelization. Indeed, the inherent parallelization potential of genetic algorithms due to the use of many independent solutions can be combined with the caching properties of cascade correlation due to the freezing of hidden layer weights to greatly simplify the simulation of the network over the entire training dataset for an entire population.

Since the weights of a hidden layer are frozen when it is added to the network, it is possible to cache the output values of the hidden layer for all training samples. Because the output of a hidden layer is connected to all subsequent layers, this is equivalent to adding a new input to be used in any subsequent training every time a hidden layer is added. Therefore, with l hidden layers, it is possible to directly compute the output of the network $\hat{y}_{i_l}$ for a training sample $(\boldsymbol{x}_i, y_i)$ using a vector $\boldsymbol{c}_{i_l},\ i \in \{1, ..., n\}$ defined as

$$\boldsymbol{c}_{i_l} = (bias, x_{i_1}, ..., x_{i_a}, v_{i_1}, ..., v_{i_l})$$

where v_{i_l} is the output value of the lth hidden layer for the ith training sample. Let m be defined as $m = a + l + 1$. With $\boldsymbol{o}_l = (w_1, ..., w_m)$, we have:

$$\hat{y}_{i_l} = \boldsymbol{c}_{i_l} \cdot \boldsymbol{o}_l$$

The output of the network for all training samples can then be computed using

$$\hat{\boldsymbol{y}}_l = \boldsymbol{C}_l \boldsymbol{o}_l$$

where $\boldsymbol{C}_l$ is a $n \times m$ matrix where each row is a vector $\boldsymbol{c}_{i_l}$ corresponding to the ith training sample. This can then be extended to compute the output of the network for all training samples using multiple different output layers:

$$\hat{\boldsymbol{Y}}_l = \boldsymbol{C}_l \boldsymbol{P}$$

In that case, $\boldsymbol{P}$ is a $m \times p$ matrix representing a population of p output layers and $\hat{\boldsymbol{Y}}_l$ is a $n \times p$ matrix containing the output values of the network for all n training samples and all p individuals. This is interesting because several high-performance parallel implementations for matrix multiplication operations already exist and these matrix multiplications are at the heart of the network construction algorithm.

5 CUDA Implementation

This work is accompanied by an open-source CUDA C++ implementation of the genetic cascade correlation algorithm called *parallel-cc*. The code comes in the form of a CUDA 6.5 project for Visual Studio 2013 and can be downloaded online [2]. This section describes some of the choices made for this implementation. In the CUDA context, *host* refers to the system board and *device* refers to the graphics board.

5.1 Data Representation

The primary data structure used throughout the project is a duplex matrix stored in row-major order. The data can be stored on host or on device, or both and can be synchronized in either direction. On device, the *cudaMallocPitch* function is used to ensure that row addresses are correctly aligned for coalescing. Data matrices are also shaped according to work parallelization so as to avoid uncoalesced access to global memory on device as much as possible.

Scalars are represented using a *real* type which can be defined as a single-precision floating point type or a double-precision floating point type. Switching between 32- and 64-bit precision is easily done by toggling a macro which affects type definitions and functions across the project.

5.2 Genetic Algorithm

A multi-population genetic algorithm is used for weight optimization. Contrary to a standard genetic algorithm, this genetic algorithm maintains a species rather than a population. A species is composed of multiple independent populations each with its own genetic parameters. The genetic parameters of each population can be set manually or generated randomly using species parameters. This presents a number of advantages. For example, rather than manually looking for interesting genetic parameters for a particular problem, specifying broad species parameters and working with more populations and lesser individuals per population can make the search more automatic. Another advantage is that it is faster to sort multiple small fitness vectors instead of a single large one. Improving spacial locality in memory reference while evolving populations is yet another example. With small populations, memory access patterns during evolution are delimited, leading to better L1/L2 cache exploitation.

The *cuRAND* API is used for random number generation. Each individual in a population has its own generator. The generator is not only used to generate random scalars but also for any stochastic operation.

The genetic operators used are roulette-wheel selection, elitism, one-point and uniform crossover, mutation and random generation. For each population,

a proportion can be specified for the use of each operator. The roulette-wheel selection operator is implemented using the roulette-wheel selection via stochastic acceptance algorithm, which provides $O(1)$ performance instead of the standard logarithmic complexity obtained with a dichotomic search and a sorted fitness vector [14].

After a species is evaluated, the populations are sorted according to fitness using the bitonic sort algorithm. The bitonic sort algorithm is very easy to parallelize and works very well for small to medium-sized arrays on a parallel architecture, a case for which it was shown to be much more efficient than other popular parallel sorting algorithms such as the radix sort found in the *Thrust* library [4].

Alternating containers are used for storing populations. This allows tracking changes from one generation to the next and makes the evolution code simpler by avoiding the use of any temporary storage.

Typically, the workload is parallelized across individuals in all major functions, meaning that individuals are handled separately, each by a different thread.

5.3 Cascade Correlation

The cascade correlation algorithm is used in conjunction with the genetic algorithm for regression problems. Data sets can be loaded from files and divided into a training set and a test set by specifying a desired test sample proportion. Test samples can either be chosen from the end of the file or randomly. The test set is used to detect overfitting, a common issue with the cascade correlation algorithm.

The training phase can be controlled in different ways. An error goal may be specified to stop the training once it is reached. A maximum number of hidden layers in the network can also be set to limit the training phase. An overfitting detection trigger may be used as well. The training stops after a specified number of hidden layers have been added without reducing the error on the test set. Alternatively, the user can manually stop the training any time. When the training is stopped, the network with the best performance on the test set is returned.

The error on a data set is measured using the sum of absolute sample errors $E = \sum |\hat{y} - y|$. The fitness function used for the output layer weight optimization is $f_o(\boldsymbol{o}) = \frac{1}{1+E}$. For hidden layer weight optimization, the fitness function is $f_h(\boldsymbol{h}) = S$. A sigmoid function, more specifically $f(x) = \frac{x}{1+|x|}$, is used as the activation function for hidden layers.

The *cuBLAS* API is used to perform matrix multiplications. Since the API works with matrices that are stored in column-major order, the provided functions cannot be used in a standard way with the matrices in this project which are stored in row-major order. A matrix multiplication $\boldsymbol{C} = \boldsymbol{AB}$ is computed by requesting that $\boldsymbol{C}^T = \boldsymbol{B}^T\boldsymbol{A}^T$ be computed instead and inverting the dimensions of each matrix. Providing a matrix $\boldsymbol{A}$ stored in row-major order as is but specifying inverted height and width causes it to be read as $\boldsymbol{A}^T$ in column-major order. This avoids unnecessary format conversion.

The amount of data transfers between host and device is minimal and the majority of the workload is executed on device, making this implementation largely dependent on device performance.

6 Experimental Results

This section presents results obtained using a GeForce GTX 460 1GB with 336 CUDA cores. It includes 2 primary experiments. The implementation is tested on the *Housing* data set [5] with varying combinations of population count and size in order to gain some insight on how using multiple independent populations affects learning performance both in terms of training time and model accuracy. The first experiment is done using fixed training and test set partitioning. In the second experiment, the test set is randomly selected every run. In both experiments, the genetic parameters are fixed and the test set accounts for 10% of the samples in the data set, or 50 samples, leaving 456 samples in the training set. The results of the first and second experiments are reported in Tables 1 and 2 and Tables 3 and 4, respectively.

For different total individual counts (i.e., population count times population size), different combinations of population size and count are tested. For each combination, the average training time, the average number of hidden layers in the final network, the MSE on the training set and the test set and the best and worst MSE achieved on the test set over 10 runs are reported. The average training time corresponds to the time it takes to build and train the network from 0 to 50 hidden layers, in seconds. The average number of hidden layers corresponds to the average number of hidden layers in the optimal network found (i.e., the network with the lowest test error). The remaining columns are self-explanatory.

In the first experiment (Tables 1 and 2), the first thing to notice is the disparity between training and test errors as well as the relatively small network size, which suggest that the seemingly subtle similarities between the training set and the test set are not captured fast enough by the algorithm. Decreasing the error on the training set increases the error on the test set in most cases, causing networks to be small. The worst case error seems to decrease as the number of populations grows for this particular test, but the large variance in

the results makes it difficult to draw any conclusion. The impact of using multiple populations on worst case performance is investigated further using auxiliary experiments described at the end of this section. The results show that there is no significant impact.

In terms of training time, the results confirm that working with multiple small populations is faster than working with a single large one when the training phase is controlled by the maximum number of hidden layers. This is best illustrated in the table with 32,768 total individuals, where the difference between the highest and lowest average training time is over 10 seconds. It is also worth noticing that while the training time decreases as the population is broken into smaller and smaller populations for the reasons mentioned in the previous section, it starts increasing after a certain point, typically when population size drops below 128. This is not surprising and is due to the block size used for the population evolution kernel. Populations are evolved in parallel using blocks of 128 threads (one thread per individual). When population size becomes lower than the block size, the workload for each block becomes less uniform, leading to lower instructions per warp (IPW) and more stalled warps. This can be easily verified by modifying the block size. Using 512 threads per block causes peak performance to shift towards a population size of 512 instead of 128.

The impact of evolving multiple populations with different configurations in a single block can be assessed using the following experiment. First, the time required to evolve 1,000 times a species made of 1024 populations of 32 individuals (32,768 total individuals) with a random generation rate of 1.0 (i.e., an entirely new population is generated each epoch, discarding the previous one) and using blocks of 128 threads (4 populations per block) is measured. In this case, the workload in a block is completely uniform and consists in generating 128 random individuals. A time of 336 ms is recorded. Next, the test is repeated with a mutation rate set to 1.0 (i.e., each generation is a mutation of the previous one). Mutation is a more costly operator than random generation. A time of 383 ms and an IPW of 3,865 (2 or more eligible warps 82% of the time in warp issue efficiency) are recorded. Then, the species mutation rate range is set to $[0.0, 1.0]$, resulting in populations with a mutation rate of different values in that range. This means that an evolution consists in a certain number of mutations and random individual generations. This results in 8 interleaved sub-blocks of mutation and random generation in each block. This time, the test completes in 523 ms with an IPW of 2,311 (2 or more eligible warps 72% of the time), thus proving that block performance can severely suffer from non-uniform workloads.

In the second experiment (Tables 3 and 4), the disparity between training error and test error is significantly reduced compared to the first experiment. This shows that the fixed training and test set partitioning used in the first experiment is particularly difficult to learn with. Randomly selected test samples lead to more consistent results and useful hidden layers. Worst-case performance however becomes largely dependent on the training and test set partitioning and

the benefits of using multiple small populations become insignificant.

Tables 5 and 6 present results similar to those obtained in the first experiment, only with a number of evolution epochs set to 50 down from 100. The impact on accuracy is small, but the training time is cut by a factor two. This is interesting because on a parallel architecture, compensating for some sequential optimization by drastically increasing the number of individuals can lead to better resource exploitation and result in lower execution time.

In order to further investigate the results obtained in the first experiment, the following auxiliary experiments are conducted. The algorithm is tested using different configurations of 1024 total individuals and 4096 total individuals. In these experiments, the test set contains 20% of the dataset samples (101 samples), leaving 405 training samples. The maximum number of hidden layers is increased to 114 and an overfitting detection trigger set to 50 hidden layers is used, meaning that training may be interrupted before reaching the maximum allowed number of hidden layers if the error on the test set cannot be reduced after adding 50 hidden layers. For the experiment with 1024 total individuals, the test set is randomly selected at each run and the algorithm is run 100 times for each different configuration. For the experiment with 4096 total individuals, 20 tests are generated by randomly partitioning the dataset into training and test sets 20 times. For each test, the training and test set are fixed and the algorithm is run 100 times for each configuration. The results of the experiments are presented in Figures 1 and 2 and Figures 3 and 4 in the form of box plots showing the minimum, first quartile, median, third quartile and maximum training error, test error, hidden layers and training time. Each box plot in the experiment with 1024 total individuals represents 100 runs each with a different training and test set partitioning, whereas in the experiment with 4096 individuals each one represents 2,000 runs figuring 20 training and test sets shared across configurations.

A few conclusions can be drawn from the plots. First, using multiple small populations instead of a single large one has little impact on accuracy. It may prevent extremely bad performance as hinted by the high maximum test error measured when using a single population compared to using multiple populations (66.08 for 1x1024 vs. 26.69 for 4x256 and 62.98 for 1x4096 vs. 34.41 for 32x128), though more tests are necessary to assert this. Another clear observation is the increasing network size as populations become smaller. Smaller populations lead to more hidden layers, in turn leading to longer training times when the network size is unbounded. Thus, although optimizing multiple small populations is faster than optimizing a single large one, they ultimately result in a longer training when the training phase is controlled by the overfitting detection trigger and not the maximum number of hidden layers because they tend to converge more slowly to an optimum.

7 Conclusion

This article explores some interesting characteristics of the genetic cascade correlation algorithm when it is implemented on a parallel architecture. By combining the caching properties of the cascade correlation algorithm with the inherent parallelization potential of genetic algorithms, the algorithm is simplified to basic operations such as matrix multiplications for which efficient solutions already exist for parallel architectures, making it easier for an efficient implementation of the algorithm to be produced.

The work includes an open-source CUDA C++ implementation of the algorithm which supports multi-population genetic algorithms to further take advantage of the target architecture. The implementation is tested on a GeForce GTX 460 1GB with 336 CUDA cores and some interesting conclusions are reached during the discussion of the results. Using multiple small populations is shown to be faster than using a single large one on a parallel architecture such as CUDA, but also to converge more slowly with each hidden layer and ultimately lead to a longer training phase when the latter is not controlled by the size of the network. In addition, using a large enough total individual count makes it possible to depend less on sequential optimization through evolution and better exploit resources in a parallel architecture.

One possible improvement to the implementation could be an alternate workload distribution for the evolution of populations. Currently, populations are divided into potentially heterogeneous blocks where different genetic operators may be used by different threads, leading to potentially lower workload uniformity within blocks and therefore potentially lower performance, especially with a very small population size. A better solution may be to group all similar operations from the different populations into the same blocks, creating only uniform blocks (i.e., mutation blocks, random generation blocks, and so on).

References

[1] Programming Guide :: CUDA Toolkit Documentation.
`http://docs.nvidia.com/cuda/cuda-c-programming-guide/`.

[2] parallel-cc.
`http://www.montefiore.ulg.ac.be/~fsafadi/parallel-cc.7z`.

[3] NVIDIA Nsight Visual Studio Edition User Guide.
`http://docs.nvidia.com/nsight-visual-studio-edition/4.1/Nsight_Visual_Studio_Edition_User_Guide.htm`.

[4] Bitonic Sort.
`http://www.montefiore.ulg.ac.be/~fsafadi/bitonic.pdf`.

[5] UCI Machine Learning Repository: Housing Data Set. `https://archive.ics.uci.edu/ml/datasets/Housing`.

[6] M. Ajtai, J. Komlós, and E. Szemerédi. An 0(n log n) sorting network. In *Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing*, pages 1–9. ACM, 1983.

[7] K. E. Batcher. Sorting networks and their applications. In *Proceedings of the 1968 AFIPS Spring Joint Computer Conference*, volume 32, pages 307–314, 1968.

[8] S. E. Fahlman and C. Lebiere. The cascade-correlation learning architecture. In *Advances in Neural Information Processing Systems 2*, pages 524–532. Morgan Kaufmann, 1990.

[9] M. Fernando and T. W.-T. Lee. Self organizing neural networks for the identification problem. In *Advances in Neural Information Processing Systems 1*, pages 57–64. Morgan Kaufmann, 1989.

[10] L. K. Hansen and M. Pedersen. Controlled growth of cascade correlation nets. In *Proceedings of the International Conference on Artificial Neural Networks*, pages 797–800, 1994.

[11] J. H. Holland. *Adaptation in Natural and Artificial Systems*. The University of Michigan Press, 1975.

[12] D. E. Knuth. *The Art of Computer Programming, Volume 3: (2Nd Ed.) Sorting and Searching*. Addison Wesley Longman Publishing Co., Inc., 1998.

[13] A. H. Kramer and A. Sangiovanni-Vincentelli. Efficient parallel learning algorithms for neural networks. In *Advances in Neural Information Processing Systems 1*, pages 40–48. Morgan Kaufmann, 1989.

[14] A. Lipowski and D. Lipowska. Roulette-wheel selection via stochastic acceptance. *Physica A: Statistical Mechanics and its Applications*, 391(6): 2193–2196, 2012.

[15] D. J. Montana and L. Davis. Training feedforward neural networks using genetic algorithms. In *Proceedings of the 11th International Joint Conference on Artificial Intelligence - Volume 1*, pages 762–767. Morgan Kaufmann, 1989.

[16] J. Moody. Fast learning in multi-resolution hierarchies. In *Advances in Neural Information Processing Systems 1*, pages 29–39. Morgan Kaufmann, 1989.

[17] M. A. Potter. A genetic cascade-correlation learning algorithm. In *International Workshop on Combinations of Genetic Algorithms and Neural Networks (COGANN-92)*, pages 123–133. IEEE Computer Society Press, 1992.

[18] I. V. Tetko and A. E. P. Villa. An enhancement of generalization ability in cascade correlation algorithm by avoidance of overfitting/overtraining problem. *Neural Processing Letters*, 6(1-2):43–50, 1997.

Population		Average				Best	Worst
Count	Size	Time	Hidden Layers	Training Error	Test Error	Test Error	Test Error
1	1024	5.7444	7.6	25.1532	15.0253	9.9296	25.3749
2	512	5.7507	12.7	23.6276	15.1427	8.87243	24.5904
4	256	5.7097	6.3	28.502	14.5934	11.0936	23.9173
8	128	5.68	19.1	23.0815	12.7926	7.72544	18.2764
16	64	5.7195	12.1	30.1349	12.6693	9.86672	19.0923
32	32	5.8223	7.1	29.7822	11.9925	8.57852	14.8908
Total		5.7378	10.8	26.7136	13.7026	9.34439	21.0237

Population		Average				Best	Worst
Count	Size	Time	Hidden Layers	Training Error	Test Error	Test Error	Test Error
1	2048	6.815	14.7	22.3864	16.1352	10.8022	22.8103
2	1024	6.802	6.2	22.5142	13.3874	11.7459	16.3759
4	512	6.7237	1	32.8006	13.7729	10.3181	18.2186
8	256	6.7126	3.3	30.4429	13.3755	10.0057	20.4157
16	128	6.6247	3.5	27.3918	12.7235	8.40769	18.0643
32	64	6.701	6	27.7456	13.1441	10.7849	16.6599
64	32	6.8261	5.1	34.3868	12.5976	10.3125	16.0724
Total		6.7436	5.7	28.2383	13.5909	10.3396	18.3739

Population		Average				Best	Worst
Count	Size	Time	Hidden Layers	Training Error	Test Error	Test Error	Test Error
1	4096	8.9534	10.7	23.1412	16.215	11.299	24.8604
2	2048	8.881	11.7	19.0772	16.1186	13.0309	23.689
4	1024	8.793	11.1	21.5032	13.2302	9.73053	16.6072
8	512	8.5829	9.4	21.5898	13.5255	9.69317	17.542
16	256	8.4892	11.7	20.3152	11.8112	9.90674	14.9905
32	128	8.432	13.1	21.9509	13.7596	10.0077	17.7071
64	64	8.5082	7.5	30.659	13.3795	11.0917	17.9532
128	32	8.7011	3.3	33.9822	13.3303	10.2261	15.5214
Total		8.6676	9.8	24.0273	13.9212	10.6232	18.6089

Population		Average				Best	Worst
Count	Size	Time	Hidden Layers	Training Error	Test Error	Test Error	Test Error
1	8192	17.6427	5.1	19.9755	14.9292	9.63952	18.7254
2	4096	17.3068	5.9	21.6052	14.1965	9.78466	18.2335
4	2048	17.0237	6.2	22.9352	12.5862	9.56828	16.0705
8	1024	16.6	5.3	27.856	13.3651	9.16044	18.3874
16	512	16.1657	12.5	23.5779	12.6474	9.56242	16.1686
32	256	15.9795	6	24.9239	11.8402	8.59518	16.1702
64	128	15.9188	7.6	22.2146	12.8953	9.18679	17.5616
128	64	16.1825	0.9	35.0585	13.4192	9.97808	15.6321
256	32	16.6815	9.4	25.1965	13.042	9.42702	16.5128
Total		16.6112	6.5	24.8159	13.2135	9.43360	17.0513

Table 1: Results obtained with fixed training and test sets (1a)

Population		Average				Best	Worst
Count	Size	Time	Hidden Layers	Training Error	Test Error	Test Error	Test Error
1	16384	33.0061	3.7	19.4949	13.8875	11.9197	16.4154
2	8192	32.0269	2.1	23.1108	14.3605	11.8449	19.632
4	4096	31.1067	4	23.1619	13.2571	10.2325	14.9378
8	2048	30.3363	7.4	20.9396	13.796	10.5962	18.701
16	1024	29.5315	5.1	22.1481	13.1562	10.5932	17.2023
32	512	28.7155	1.6	26.0216	14.1073	11.5454	17.1557
64	256	28.3895	11.8	20.7006	14.533	8.26138	20.742
128	128	28.4287	1.1	28.7517	12.7224	7.37068	18.1893
256	64	28.9285	4.5	26.0835	11.8389	9.51295	16.0983
512	32	29.6812	6.2	28.3104	12.7063	9.57904	16.5264
Total		30.0151	4.8	23.8723	13.4365	10.1456	17.5600

Population		Average				Best	Worst
Count	Size	Time	Hidden Layers	Training Error	Test Error	Test Error	Test Error
1	32768	65.7259	13	15.1833	13.4346	10.6843	19.8109
2	16384	63.6212	6.2	21.4143	13.337	7.26945	17.5696
4	8192	61.6337	6.9	22.4735	13.9666	11.9585	18.3517
8	4096	59.4162	3.5	20.8499	13.502	11.4353	18.0738
16	2048	57.895	12.5	16.1652	13.3681	10.7183	15.6923
32	1024	56.0563	21.9	13.3399	11.5464	9.78405	14.0714
64	512	54.4305	8.9	21.3979	13.6064	10.3957	19.9792
128	256	53.8164	7.9	21.97	13.5777	12.029	16.6165
256	128	53.921	6.6	24.2377	12.535	10.9093	14.3517
512	64	54.852	1.9	28.563	11.3465	8.51013	13.8668
1024	32	56.5238	6.5	26.2171	12.7646	9.13264	15.7992
Total		57.9902	8.7	21.0738	12.9986	10.2570	16.7439

Table 2: Results obtained with fixed training and test sets (1b)

Population		Average				Best	Worst
Count	Size	Time	Hidden Layers	Training Error	Test Error	Test Error	Test Error
1	1024	5.7866	30.7	8.79973	16.2081	8.68358	27.9637
2	512	5.7828	21.9	11.1445	18.3999	9.82574	37.3507
4	256	5.7254	26.5	10.201	15.1386	8.24654	43.3569
8	128	5.6837	36.8	8.08679	16.6551	8.91677	27.4543
16	64	5.7254	33	11.7211	16.2391	7.25801	25.5675
32	32	5.828	36.7	12.7327	21.2692	8.8733	30.1395
Total		5.7553	30.9	10.4476	17.3183	8.63399	31.9721

Population		Average				Best	Worst
Count	Size	Time	Hidden Layers	Training Error	Test Error	Test Error	Test Error
1	2048	6.8076	20.6	10.4119	12.5716	7.24871	16.4769
2	1024	6.8415	17.2	12.2465	13.626	6.70158	19.0794
4	512	6.7526	28.9	7.16465	13.8171	7.7502	22.039
8	256	6.7058	33.7	8.61757	14.218	7.87063	21.7652
16	128	6.6317	37.3	7.8377	14.3334	5.83973	21.2703
32	64	6.7047	28.6	14.9117	11.175	8.02507	16.5312
64	32	6.834	40.4	11.3318	17.3591	5.79048	49.7445
Total		6.7540	29.5	10.3603	13.8715	7.03234	23.8438

Population		Average				Best	Worst
Count	Size	Time	Hidden Layers	Training Error	Test Error	Test Error	Test Error
1	4096	8.9666	23.7	7.90538	14.1123	7.90678	23.6956
2	2048	8.8635	17.6	12.5183	15.6521	9.55615	28.2149
4	1024	8.7665	16.4	11.3872	16.5327	8.8409	32.594
8	512	8.5904	28	9.21454	12.4991	5.73309	41.0194
16	256	8.5073	21.4	13.1608	12.281	6.05751	27.9876
32	128	8.4413	32.3	9.04951	12.8213	4.81874	25.843
64	64	8.5229	40.7	8.68574	14.544	9.95593	22.6624
128	32	8.7021	31.7	13.859	17.9074	10.9365	29.5512
Total		8.6701	26.5	10.7226	14.5437	7.9757	28.9460

Population		Average				Best	Worst
Count	Size	Time	Hidden Layers	Training Error	Test Error	Test Error	Test Error
1	8192	17.6891	19.1	9.97156	14.9819	6.33594	24.8311
2	4096	17.3195	29.1	6.09559	14.319	9.38157	22.1386
4	2048	17.0635	22.8	10.4267	15.3268	7.57837	31.6053
8	1024	16.5957	24.7	7.84281	14.2335	7.81084	29.1284
16	512	16.1536	18.3	11.8615	13.2321	9.09361	19.4294
32	256	15.9389	24.6	11.5548	14.3233	9.23231	20.5262
64	128	15.9407	34.5	8.09448	16.4088	7.15903	29.7323
128	64	16.2059	28.5	11.5789	16.6488	4.93472	41.7021
256	32	16.6839	38.6	11.3399	13.1785	6.87062	21.7681
Total		16.6212	26.7	9.86292	14.7392	7.59967	26.7624

Table 3: Results obtained with random training and test sets (a)

Population		Average				Best	Worst
Count	Size	Time	Hidden Layers	Training Error	Test Error	Test Error	Test Error
1	16384	33.1539	20.8	8.28689	11.4533	6.21434	19.3758
2	8192	32.2125	20.5	7.07369	14.8537	8.1422	25.9423
4	4096	31.2328	26.1	6.22508	17.0958	9.86969	30.6308
8	2048	30.3883	26	9.30055	16.056	7.53701	31.4495
16	1024	29.5724	23.6	9.06147	15.0136	9.94383	24.0633
32	512	28.7664	20.3	8.83704	15.2959	7.6053	29.8551
64	256	28.4327	31.7	9.80965	15.5661	6.74968	39.2089
128	128	28.4607	16.3	13.8391	13.1809	7.2251	25.9738
256	64	28.9291	33.5	10.1962	14.4106	7.06882	24.2269
512	32	29.71	36.9	12.1729	15.4333	9.55274	22.3314
Total		30.0859	25.6	9.48026	14.8359	7.99087	27.3058

Population		Average				Best	Worst
Count	Size	Time	Hidden Layers	Training Error	Test Error	Test Error	Test Error
1	32768	65.0992	20.2	7.68169	11.9521	5.41838	29.6517
2	16384	63.4841	19.8	8.86801	11.314	6.68024	20.0277
4	8192	61.5051	27.7	7.16731	16.9692	8.25641	26.3433
8	4096	59.3211	19.7	7.00603	12.082	6.01189	22.2692
16	2048	57.8055	29.8	6.81576	13.3609	7.93704	27.4768
32	1024	56.0592	18.8	9.34539	16.4611	10.1129	23.6008
64	512	54.4021	34.5	5.74977	17.5853	7.00735	42.7785
128	256	53.8582	21.5	10.8576	19.832	10.0531	36.1215
256	128	53.8308	28.7	11.6495	18.6566	10.6928	31.2416
512	64	54.8088	32.9	10.2964	14.5821	7.13799	20.7592
1024	32	56.4425	39.4	10.3247	19.6797	9.96527	40.9079
Total		57.8742	26.6	8.70565	15.6795	8.11576	29.1980

Table 4: Results obtained with random training and test sets (b)

Population		Average				Best	Worst
Count	Size	Time	Hidden Layers	Training Error	Test Error	Test Error	Test Error
1	1024	2.9376	10.4	28.6406	17.7922	10.6933	31.748
2	512	2.945	10.8	24.1731	16.11	10.2608	22.4682
4	256	2.9302	4.3	35.6086	14.1943	6.91624	22.6771
8	128	2.914	8.7	38.0709	13.0964	9.23085	16.6226
16	64	2.9323	11	27.5158	14.3576	11.1315	16.1959
32	32	2.9851	8	38.0299	14.1412	6.82335	18.9557
Total		2.9407	8.9	32.0065	14.9486	9.17601	21.4446

Population		Average				Best	Worst
Count	Size	Time	Hidden Layers	Training Error	Test Error	Test Error	Test Error
1	2048	3.5058	7.3	25.397	16.4004	11.0134	32.0249
2	1024	3.4844	11.8	20.8501	16.4594	10.2645	25.913
4	512	3.4567	6.4	29.4426	14.7056	10.1829	20.1048
8	256	3.436	6.1	32.4958	14.3556	11.441	17.428
16	128	3.3905	7.3	30.717	13.4743	7.68237	24.1688
32	64	3.4323	9.9	26.9228	13.0089	10.4453	17.975
64	32	3.4926	15.7	32.8819	12.3857	8.16403	19.1199
Total		3.4569	9.2	28.3867	14.3986	9.88479	22.3906

Population		Average				Best	Worst
Count	Size	Time	Hidden Layers	Training Error	Test Error	Test Error	Test Error
1	4096	4.6091	15.6	21.885	15.0998	10.697	28.4413
2	2048	4.5177	9.7	25.5822	13.2116	11.4431	16.712
4	1024	4.4892	5	28.1588	15.1242	10.7369	18.7719
8	512	4.3733	6.2	28.0812	13.4197	7.87407	16.3264
16	256	4.3272	4.3	26.8258	14.0732	9.19711	21.4063
32	128	4.2987	17.7	23.4574	13.2223	10.6287	15.7383
64	64	4.3389	9.2	24.103	13.0291	10.0032	16.2899
128	32	4.4258	15.1	29.1083	12.8179	9.01312	16.7183
Total		4.4225	10.4	25.9002	13.7497	9.94915	18.8006

Population		Average				Best	Worst
Count	Size	Time	Hidden Layers	Training Error	Test Error	Test Error	Test Error
1	8192	8.9584	15.6	19.9823	15.854	12.5602	24.8782
2	4096	8.7448	14.8	19.3946	13.254	10.8309	16.7977
4	2048	8.6227	10.4	23.4762	14.2682	9.15028	21.0271
8	1024	8.4192	7.8	23.9407	11.17	9.45436	12.9457
16	512	8.2094	10.9	21.0673	12.1152	10.0815	15.6551
32	256	8.0856	15.7	25.2934	12.2781	10.2268	15.0444
64	128	8.0576	3	32.8097	12.0329	9.82625	16.1738
128	64	8.1955	14.8	24.2395	12.9621	9.34046	18.4505
256	32	8.4579	6.8	30.0937	12.4577	8.5789	15.4924
Total		8.4168	11.1	24.4775	12.9325	10.0055	17.3850

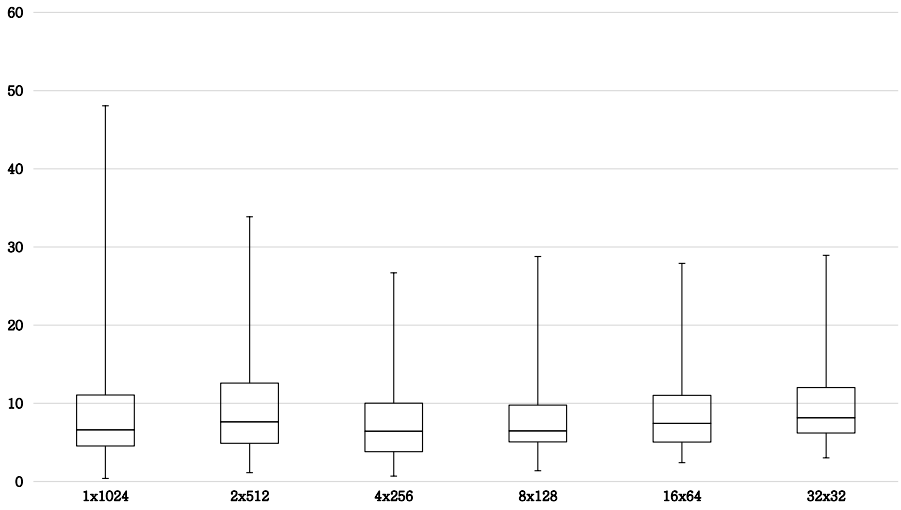
Table 5: Results obtained with fixed training and test sets (2a)

Population		Average				Best	Worst
Count	Size	Time	Hidden Layers	Training Error	Test Error	Test Error	Test Error
1	16384	16.7614	3.5	26.8023	14.283	10.7267	19.9507
2	8192	16.1715	16.1	15.1361	12.996	10.7763	15.6309
4	4096	15.7204	8	19.978	13.894	7.93584	15.8702
8	2048	15.3757	4.3	26.4008	14.3963	11.1337	18.9091
16	1024	14.9467	6.5	24.0266	13.4192	12.2344	15.4028
32	512	14.5467	11.9	21.3798	11.9368	8.83372	13.867
64	256	14.409	6.1	25.3503	14.2053	10.7836	19.6191
128	128	14.3923	14.5	21.8821	14.18	9.35829	18.7834
256	64	14.6231	4.7	28.4069	12.5931	7.78428	20.2294
512	32	15.0278	10.3	29.741	12.0001	9.44003	14.3281
Total		15.1975	8.6	23.9104	13.3904	9.90069	17.2591

Population		Average				Best	Worst
Count	Size	Time	Hidden Layers	Training Error	Test Error	Test Error	Test Error
1	32768	33.027	11	20.2192	14.0776	10.1778	19.2708
2	16384	31.9724	7.9	23.0951	13.9906	10.5227	18.3219
4	8192	31.0466	6.9	23.2507	13.7711	10.6745	17.4305
8	4096	29.9824	9.7	16.0172	11.8862	8.48604	14.352
16	2048	29.1772	10.6	20.5569	13.6273	10.8444	18.0473
32	1024	28.2283	6.7	21.7664	13.9419	12.801	16.1334
64	512	27.3834	10.5	21.7159	13.6081	9.17518	17.8265
128	256	27.1135	7	24.2265	13.2249	9.70003	16.5823
256	128	27.1462	4	28.3921	12.5484	8.66757	15.8086
512	64	27.6345	10.7	22.9369	13.4483	8.23694	18.0201
1024	32	28.4831	3.6	31.5642	12.4989	9.59946	17.2554
Total		29.1995	8.1	23.0674	13.3294	9.89869	17.1863

Table 6: Results obtained with fixed training and test sets (2b)

(a) MSE on the training set



(b) MSE on the test set

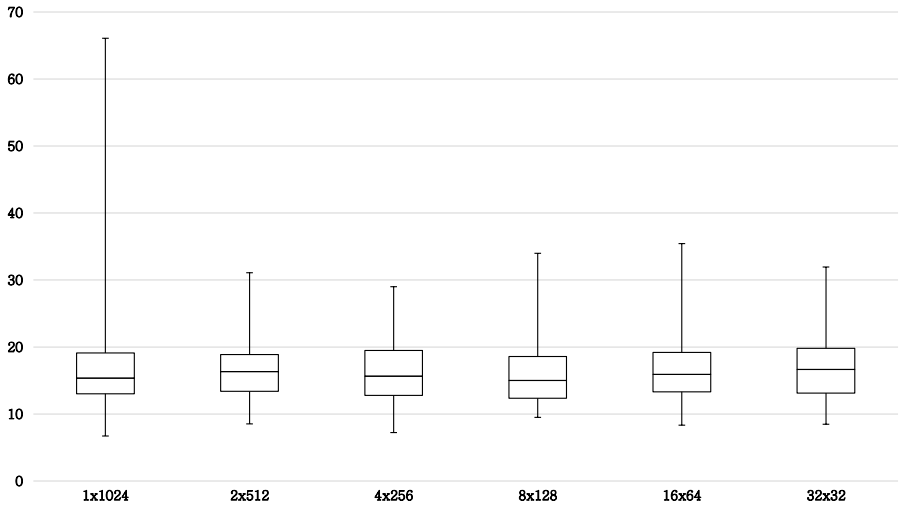
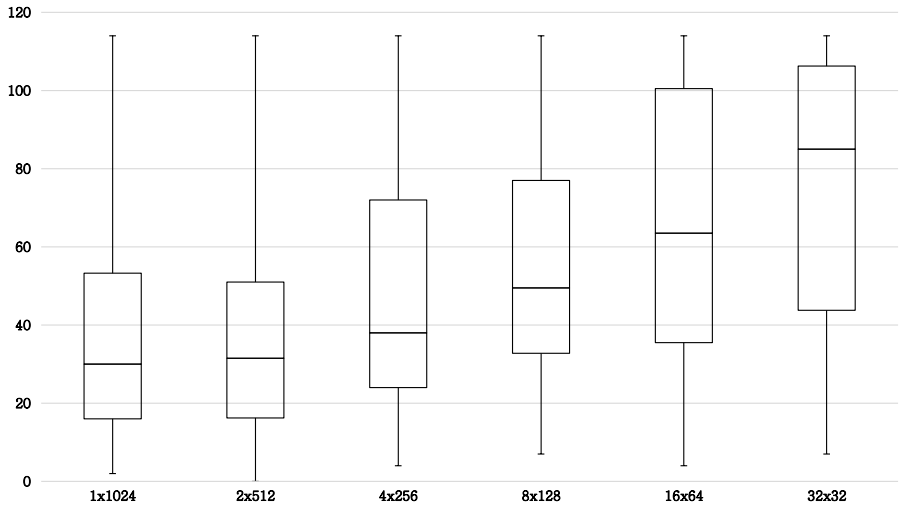


Figure 1: Training and test error versus population configuration (1a)

(a) Hidden layers



(b) Training time in seconds

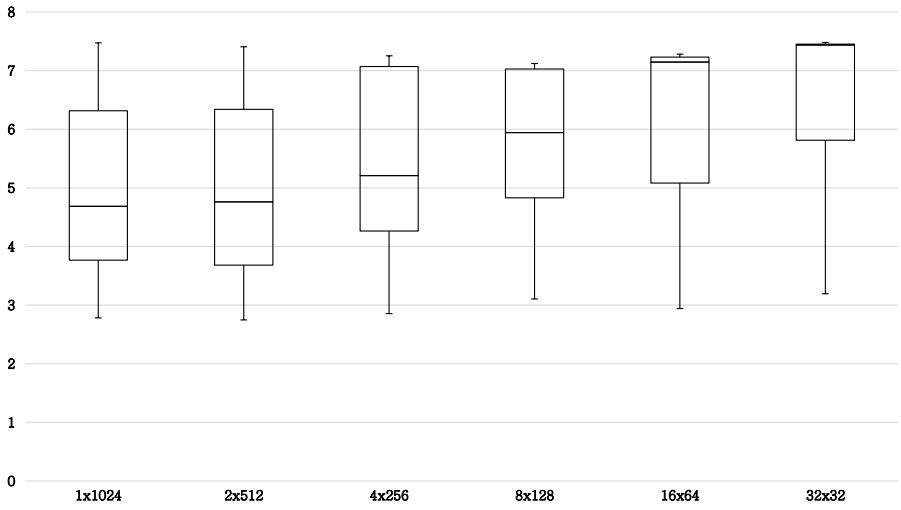
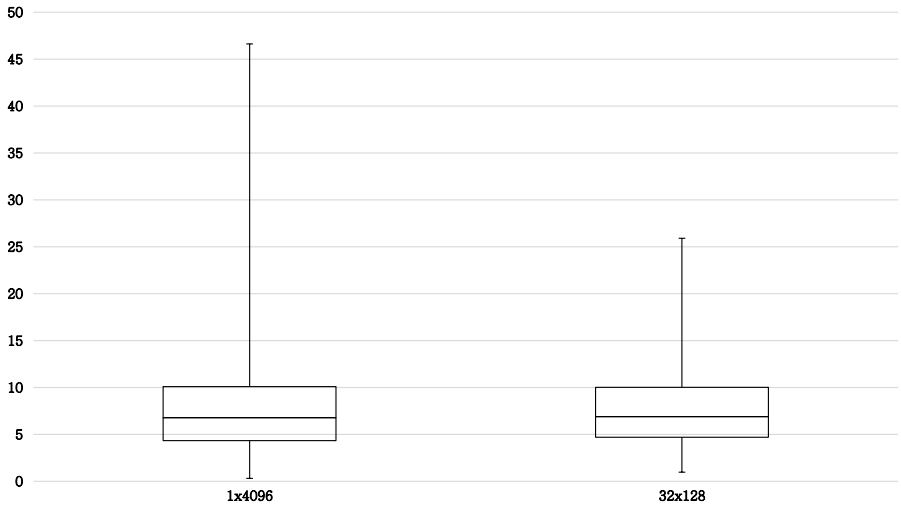


Figure 2: Training time and hidden layers versus population configuration (1b)

(a) MSE on the training set



(b) MSE on the test set

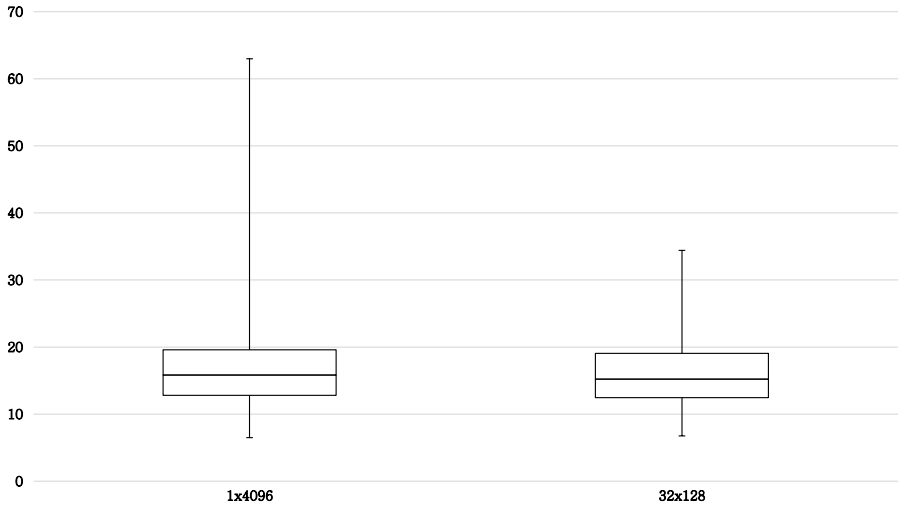
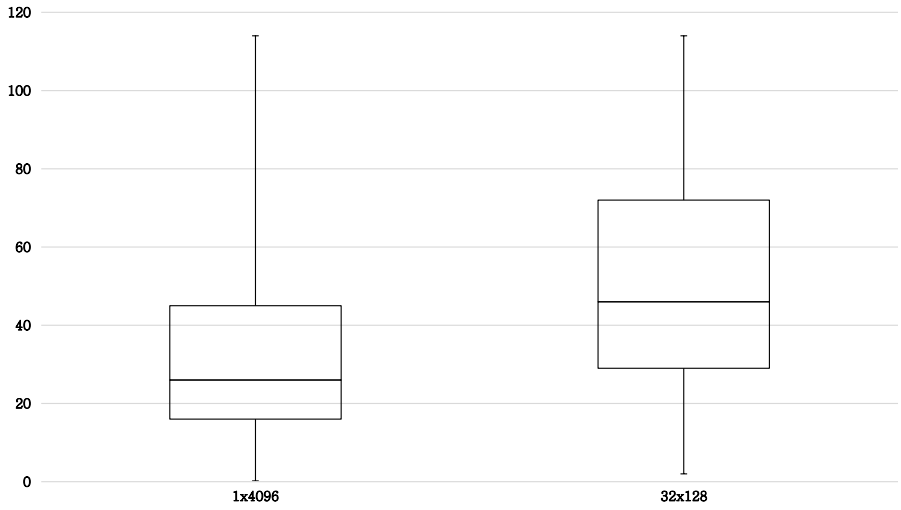


Figure 3: Training and test error versus population configuration (2a)

(a) Hidden layers



(b) Training time in seconds

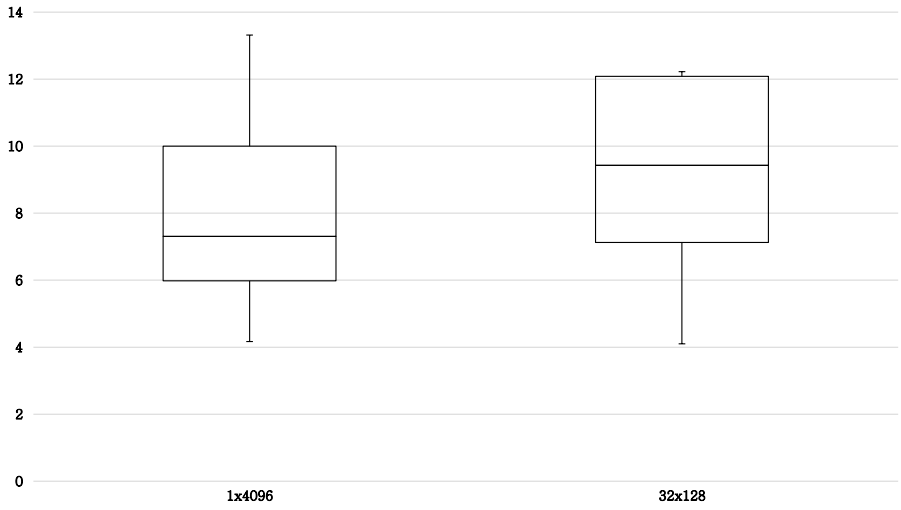


Figure 4: Training time and hidden layers versus population configuration (2b)